林 晴比古
実用マスターシリーズ

C言語 クイック入門＆リファレンス

林 晴比古

SoftBank
Creative

林晴比古
実用マスターシリーズ

●●●

新訂 新C言語入門 ビギナー編

新訂 新C言語入門 シニア編

新訂 新C言語入門 スーパービギナー編

●

明快入門C++ ビギナー編

明快入門C++ シニア編

●

改訂 新Java言語入門 ビギナー編

改訂 新Java言語入門 シニア編

●

明快入門Visual Basic 2008ビギナー編

明快入門Visual Basic 2008シニア編

●

明快入門Visual C++ 2008ビギナー編

明快入門Visual C++ 2008シニア編

●

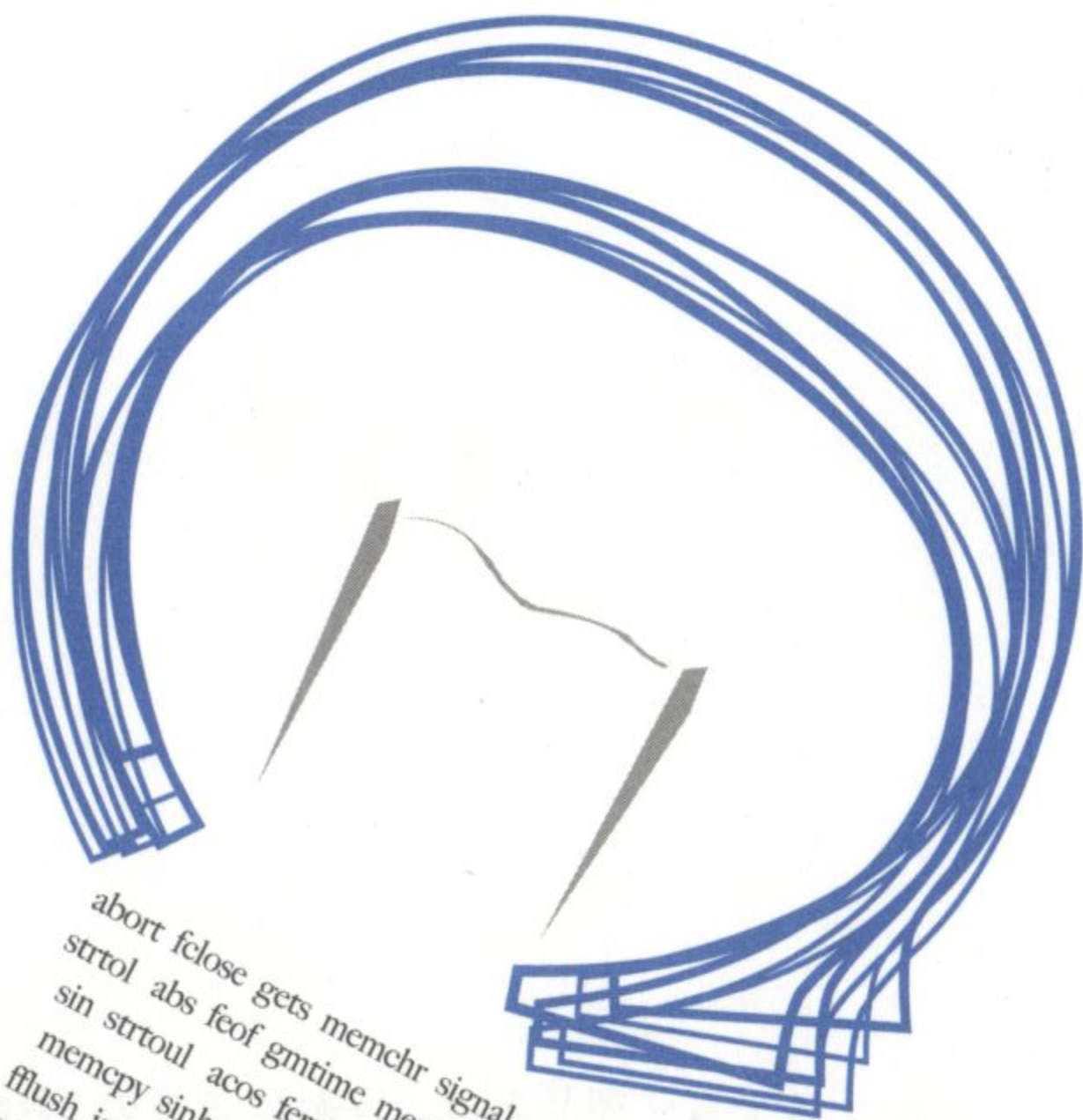
明快入門SQL

高級言語プログラマのためのアセンブラ言語入門

●

改訂 新Linux/UNIX入門

●

明快入門 コンパイラ・インタプリタ開発

カバーデザイン/三部 智也（Claro）

C言語
クイック入門&
リファレンス

林 晴比古

●お知らせ

本書に関する情報などは、インターネットの次のURLアドレスに登録される予定になっていますので、ご利用ください。

http://www.sbcr.jp/books/hayashi/

まえがき

本書はプログラミング言語Cについて解説したものです。「C言語の項目を網羅していて、しかしコンパクト」というのが本書の特徴です。従来のC規格だけでなく、新しいC99機能についても十分にサポートしています。それぞれの説明は見開き方式でレイアウトし、項目がすっきりと並び、ポイントがすんなりと理解できるようになっています。

現在の恵まれたコンピュータ環境においては、さまざまなプログラミング言語が利用可能です。活動範囲の広いプログラマは、多くの開発言語を使いこなしています。たとえば、C、C++、Java、Visual Basic、C#、Perl、Ruby、PHPといったものを使い分けます。昨日はJavaを使い、今日はC++を使う、ということもあります。

このような場合、困るのは言語文法が混乱してしまうことです。たとえばCとC++で混乱します。

自分でそのような経験をし、また周囲から同じ声を聞いていたので、しだいに「ある言語の文法や機能を楽に確認できる薄い本があればいいのではないか」と考えるようになりました。

本書はそのような経緯で、執筆された本です。C言語の文法や用法をすばやく確認できるように構成されています。またC知識が曖昧になっているとき、本書を読むことで、知識の再構築、つまり「クイック入門」ができるようになっています。

本書の内容は次のとおりです。

第1章から第18章までは、C言語規格にもとづいて解説します。説明はできるだけコンパクトに、ポイントを中心に解説します。見開きレイアウトなので、ひとつの項目の説明が長ながと続くことはありません。知りたい項目が、すぐに見つかるように工夫されています。

第19章は、C規格として備わっている全ライブラリ内容の要約を示します。C99規格に用意されている24個のライブラリについて、そこに登場するすべての識別子(型、マクロ、プラグマ、関数)の説明をしています。そして最後の第20章は、C言語の主要関数について「形式」「戻値」「説明」「用例」方式で解説を行ないます。この章を見ると、ライブラリ関数の実際の用法が分かります。

C言語は従来規格と、後で追加されたC99規格には、大きな違いがあります。C99規格は、どちらかというと、高度で複雑で特殊な項目が多く、通常のプログラミングに用いられる機会は多くありません。サポートの不十分なC処理系もあります。

本書はC99規格についても十分な解説を加えましたが、従来規格との混乱を避けるために、C99というマークを採用しました。入門者はC99マークの付いた箇所については、とりあえず後回しにしてもよいでしょう。

本書は、いつも手元に置き、ハンドブック的な感覚で早引き利用するのに適しています。必要なとき必要な知識が手にはいる、という使い方をしていただければ幸いです。

2010年6月6日　林 晴比古

CONTENTS

第11章　関数 ······ 115

第12章　構造体型 ······ 139

第13章　ビットフィールド ······ 151

note

第1章

Cの基本的な知識

C Quick Reference

001 基本機能1

Cの基本的な知識1

小さなプログラム

ここではまずC言語のもっている雑多な規則を説明します。まとまった説明の必要な規則については次章から整理して示します。それ以外の、トピックス的な内容についてここでのべておきます。

はじめに小さなプログラムをひとつ提示し、その意味を説明します。次のプログラムを実行すると「Hello」と表示されます。

```
/* 小さなプログラム*/
#include <stdio.h>

void message(void)
{
    printf("Hello¥n");
}

int main(void)
{
    message();
    return 0;
}
```

/* ～ */の部分はコメントで、プログラムとしては無視されます。「#include <stdio.h>」は、このプログラムのための情報を取り込んでいます。

message()とmain()は関数です。Cプログラムはmainという名前の関数から実行開始されます。それ以外の関数は、必要に応じて記述します。

main()関数の中では、message()関数を呼び出しています。printf()は、はじめから用意されている関数(ライブラリ関数)で、データを出力する機能をもちます。

「return 0;」は呼び出し側に値0を戻します。main()関数はこのreturn文で値を戻して終了します。この例は小さくて単純ですが、Cプログラムの基本的なルールが表明されています。

予約語

Cでは次のものを**予約語（キーワード）**にしています。予約語は識別子には使えません。

予約語一覧

```
auto     continue  enum    if        short    switch   volatile
break    default   extern  int       signed   typedef  while
case     do        float   long      sizeof   union
char     double    for     register  static   unsigned
const    else      goto    return    struct   void
```

注：C99では次の予約語が追加された。

```
inline   restrict  _Bool   _Complex  _Imaginary
```

識別子

識別子に用いることのできる文字は次のとおりです。識別子は英文字または_（下線）ではじまります。大文字と小文字は区別されます。識別子の長さは先頭から最低でも31文字（C99では63文字）が識別されます。さらにC99規格では、**国際文字名**を使うことができます。

識別子に使用できる文字

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
_ 0 1 2 3 4 5 6 7 8 9
```

予約済み識別子

上に示した予約語は識別子には使えませんが、次のルールの名前も**予約済み識別子**とされています。処理系の内部的な用途、あるいは今後の拡張として使われる可能性があります。

(1) 下線に続き大文字1字ではじまる識別子、およびふたつの下線ではじまるすべての識別子は常に予約済みとする（**_Name**や**__name**の形式）
(2) ひとつの下線ではじまるすべての識別子は、「通常の識別子の名前空間およびタグの名前空間におけるファイル有効範囲をもつ識別子」としては常に予約済みとする

あらかじめ定義された識別子

C99ではあらかじめ定義された識別子として__func__が用意されました。関数定義時にブロック開始{の直後に、この識別子が暗黙のうちに宣言されます。この__func__の内容は現在の関数名になります。

```
void myfunc(void)                              // 任意の関数
{
    static const char __func__[] = "myfunc";   // この暗黙の宣言が入る
    ...
}
```

利用例

```
printf("%s¥n", __func__);                      // 現在の関数名を表示
```

コメント(注釈)

コメントは次のように書きます。/*～*/は範囲指定スタイルで、複数行にわたってもかまいません。/*～*/はひとつの空白とみなされます。また//があると行末までコメントです。これはC99規格で用意されましたが、それ以前の多くの処理系で先行して採用されていました。

```
/* この範囲はコメント */

/*
    この
    範囲は
    コメント
*/

// 行末までコメント
```

字下げと連続空白

Cでは連続した空白は、ひとつの空白と同じです。プログラムは適切に**字下げ(インデント)**すると読みやすくなります。

字下げの例

```
void foo(void)
{
    int i;
    for (i=1; i<=5; i++) {
        printf("%d¥n", i);
    }
}
```

字句

字句(token) はCプログラム解釈上の、最小構成単位になるものです。規則にしたがって文字を集めたものが字句になります。トークンともいいます。字句には次のものがあります。

字句

キーワード	if for long など
識別子	変数名や関数名など
定数	100　12.345 など
文字列リテラル	"abc" など
区切り子	[ ] ( ) { } ++ -- + - など

例：「a=dt[pos]+100;」は次の字句として処理される

```
a  =  dt  [  pos  ]  +  100  ;
```

オブジェクト

オブジェクトという言葉は広範囲に用いられています。たとえばオブジェクトコードというと機械語の形式になったコードのことをいいます。言語処理系の世界で単にオブジェクトというときは通常、

――データを書き込んだり、参照したりできる、名前のついた記憶領域のこと

を指しています。アドレスはオブジェクトのある場所をいい、変数はオブジェクトを識別する名前です。

また特殊なものとして、アドレスや名前のないオブジェクトや、書き込みできないconstオブジェクトなどもあります。

002 基本機能2

Cの基本的な知識2

逆斜線文字

Cでは改行文字を\nと表現します。しかしこの**逆斜線文字**\はJISキーボードには割り当てられていません。そのためJISキーボードの場合は(日本のPCの場合は)、同じ文字コード(0x5C)である'¥'を使います。したがって改行文字は¥nとなります。本書でも'¥'を使用します。

行の継続

Cプログラムは「#ではじまる前処理指令」と「//によるコメント」は行末を意識した処理をします。それ以外の場所では「空白を入れることのできる位置」で、自由に行分けすることができます。

また継続指定文字¥を行末に置くと、任意の位置で(語の切れ目でも)行分けが可能になります。Cコンパイラは、

——行末に'¥'があったら、その'¥'自身と、続く改行文字をないものとみなす

という判断をすることで2行を継続させます。改行後の先行空白も意味をもつので注意してください。次の例は前処理記述を行分けしています。

標準の書き方

```
#define putd(n) printf("%d¥n", n)
```

¥による行分け例

```
#define putd(n) ¥
printf("%d¥n", n)
```

式と式文

式とは、定数や変数そのもの、またそれらを演算子で結合したもの(演算子とオペランドの並び)をいいます。関数呼び出しも式にふくまれます。次のものは式です。a, b, cは適切な変数とします。

式の例

a	a = 100
10	a > 2000
++c	"abcdefg"
a + b	printf("Hello¥n")

式にセミコロンをつけると文になります。これは特に**式文**と呼ばれます。たとえばa; 10; ++c;はどれも式文です。

定数式

定数式は、「コンパイル時に定数として評価できる式」のことをいいます。たとえば10や10+20は定数式です。一方、aやa+30は式ではあるけれど定数式ではありません(aは変数とする)。定数式という用語は、

——静的変数の初期設定時の初期値は定数式でなければならない

というように用いられます。

文

文はプログラム実行のための最小構成単位です。文は逐次実行され、実行されることでなんらかの効果を残します。文についての規則をまとめたのが文法です。文は次のものから構成されます。

文

ラベルつき文	
式文	
複合文	{}
選択文	if switch
繰り返し文	for while do-while
分岐文	goto continue break return

ひとつの文

Cではif, whileなどの**制御文**は、「後続するひとつの文」を制御します。このとき「ひとつの文」には次のようなバリエーションがあります。

単純文

```
a = 10;
```

複合文

```
{a = 10; b = 20;}
```

カンマ演算子で結合された文

```
a = 10, b = 20;
```

式の値

式を実行させるとなんらかの値を発生します。これを**式の値**といいます。たとえば、

```
if (a > 100)
```

を実行すると「a>10」という式が「式の値」を発生し、その値にもとづいて選択処理が行なわれます。

またCでは、代入処理も式なので、式の値をもちます。この機能を用いると、コンパクトな記述が可能になります。次の例は「ch = getchar()」が式の値をもち、それがEOFでないことをチェックしています。

```
while ((ch = getchar()) != EOF) {
    ...
}
```

もし「式の値」という機能がなければ、これは次のように書かれるところです。

```
ch = getchar();          ——最初の判定のためのch設定
while (ch != EOF) {
    ...
    ch = getchar();      ——2回目以降のループ判定用のch設定
}
```

3文字表記

3文字表記は[ ] { } ¥ | ^ ~ #という文字セットが使えない環境でも、Cプログラミングを可能にするもので、次のものがあります。左のように書くことで、右の文字とみなされます。

3文字表記

3文字表記	該当文字
??(	[
??)	]
??<	{
??>	}

3文字表記	該当文字
??/	¥
??!	|
??'	^
??-	~

3文字表記	該当文字
??=	#

ISO646代替つづり機能

3文字表記と同様に、文字セットに不足のある環境のために**ISO646代替つづり機能**が用意されています。これはiso646.hヘッダで提供されていて、その内容は次のとおりです。

ISO646代替つづり機能の定義

#define and	&&
#define and_eq	&=
#define bitand	&
#define bitor	|
#define compl	~
#define not	!
#define not_eq	!=

#define or	||
#define or_eq	|=
#define xor	^
#define xor_eq	^=

記述例

```
if (a==10 and b==20) c = 30;
              └── &&の意味になる
```

003 C言語の規格

Cの基本的な知識3

C言語規格の流れ

C言語の規格は国際規格番号であるISO/IEC 9899で定義されています。その国際規格は適時、日本語化されてJIS規格として翻訳発行されています。これまで日本では次の3つのJIS規格が発行されています。

(1)プログラミング言語C　JIS X 3010:1993
　対応する国際規格　ISO/IEC 9899:1990
　(元になる米国規格)　(ANSI X3.159-1989)
(2)プログラミング言語C(追補1)　JIS X 3010:1996
　対応する国際規格　ISO/IEC 9899:1990 Amendment 1:1995
(3)プログラミング言語C　JIS X 3010:2003
　対応する国際規格　ISO/IEC 9899:1999

これらの規格は制定した年号からC89 C95 C99などと略称で呼ばれています。C95については「C89(追補1)」または「C89(Amendment 1)」と呼ばれることもあります。

C99規格で追加された機能は、大まかにいって次のものになります。

(1)規格から外れる機能ではあるが一般的に使われていた機能を認める
　例：`//`コメント機能、`%lf`変換指定子
(2)C++などで使われている機能で、Cでも採用が期待されていた機能を導入
　例：インライン関数、`for`文第1項での変数宣言機能
(3)その他の新機能を追加。また微細な機能修正
　例：新しいデータ型、複素数機能のサポート

この中で(3)が改定の中心であり、それに関連して追加された新関数の数も膨大になります。

C99で改正された主な変更点を次に示します。これは新しいJIS規格書の「JIS X 3010:2003(ISO/IEC 9899:1999) プログラミング言語C」で示されている内容を引用したものです(番号は著者が付与)。主要な変更ポイントについては、本文のそれぞれの章で説明します。

C言語規格の主な変更点(JIS規格書より抜粋)

(01) 2文字表記及び`<iso646.h>`での限定された文字集合のサポート

(02) <wchar.h>及び<wctype.h>でのワイド文字ライブラリのサポート
(03) 有効型による従来以上に厳密な別名付け(aliasing)の規則の導入
(04) restrict修飾ポインタ
(05) 可変長配列
(06) フレキシブル配列メンバ
(07) 仮引数における配列型の宣言子でのstatic及び型修飾子の指定
(08) <complex.h>での複素数(及び虚数)のサポート
(09) <tgmath.h>での型総称マクロ
(10) long long int型及びライブラリ関数
(11) 翻訳限界での最小値の拡張
(12) <float.h>での浮動小数点の特性の追加
(13) 暗黙のint型宣言の排除
(14) 正確な整数除算
(15) 国際文字名(\u及び\U)
(16) 識別子の拡張
(17) 16進浮動小数点定数, 並びにprintf/scanf変換指定子%a及び%A
(18) 複合リテラル
(19) 初期化における要素指示子
(20) //注釈
(21) <inttypes.h>及び<stdint.h>での拡張整数型及びライブラリ関数
(22) 暗黙の関数宣言の排除
(23) 前処理時におけるintmax_t/uintmax_t型での計算の実行
(24) 宣言部分とコード部分の混合
(25) 選択文及び繰返し文に対する新しいブロック有効範囲
(26) 整数定数型の規則
(27) 整数拡張の規則
(28) 可変個の実引数をもつマクロ
(29) <stdio.h>及び<wchar.h>でのvscanf関数群
(30) <math.h>での数学関数の追加
(31) <fenv.h>での浮動小数点環境へのアクセス
(32) IEC 60559(IEC 559又はIEEE算術ともいう)のサポート
(33) 列挙型の宣言における最後のコンマの許容
(34) printfでの%lf変換指定子の許容
(35) インライン関数
(36) <stdio.h>でのsnprintf関数群
(37) <stdbool.h>での論理型
(38) 同じ型修飾子の複数回出現の許容
(39) 空のマクロ実引数
(40) 構造体型の新しい互換性規則(タグの互換性)
(41) あらかじめ定義されたマクロ名の追加
(42) _Pragma前処理演算子
(43) #pragma STDCの導入
(44) あらかじめ定義された識別子__func__
(45) VA_COPYマクロ
(46) strftime変換指定子の追加
(47) LIA(言語共通算術)互換の附属書
(48) バイナリファイルの先頭におけるungetcを廃止予定とする
(49) 配列の仮引数の別名付けを廃止予定としていたことの排除
(50) 配列のポインタ型への変換を左辺値に限定しない
(51) 集成体及び共用体の初期化における制約の緩和
(52) 移植性のあるヘッダ名に関する制約の緩和
(53) 値を返す関数のreturn文に式を書かないことを許さない(逆も同じ)

004 処理系のC99規格対応

Cの基本的な知識4

処理系のC99規格対応

現在流通しているC処理系はC89規格に対応しています。また多くの処理系はC95(=C89(Amendment 1))規格にも対応しています。

これらのC規格の範囲内でプログラムを作成すれば、他の処理系とソースコードを共有できます。

一方、C99規格は、その変更があまりにも大きいために、対応はまちまちです。Visual C++は、限定的な対応になっています。GNU Cは対応が進んでいますが、いろいろと癖のある動作などがあります。また仮に処理系がC99に対応していても、「-std=c99」のようなオプションの指定が必要な場合もあります。C99機能を使うときは、その準拠度合いと実動作を調査しながら使用することが必要です。

現状では可搬性のあるプログラムを作成するなら、C99機能に依存しないコード作成が安心です。なお1行コメント機能(//)のように、広く普及しているものもあります。このような機能は使ってかまいません。

第2章

定数

C Quick Reference

005 定数の種類

定数1

定数は定まった数を表現するもので、次のものがあります。

整数定数　浮動小数点定数　文字定数　列挙定数

また文字列を表現するものとして、

文字列リテラル

があります。列挙定数については「018 列挙型データ」(p. 26)で説明します。

006 整数定数

定数2

整数定数には8進数、10進数、16進数の各表現があります。また接尾語U L LL を数字の末尾につけることで、定数の大きさと符号有無を指定できます。

整数定数の表現

区別	説明	例
8進定数	0～7の数字で構成される 0ではじまる	**033**
10進定数	0～9の数字で構成される 0以外の数字ではじまる	**1234**
16進定数	0～9, A～F(a～f) で構成される "**0x**"または"**0X**"ではじまる	**0x1a** **0XAB03**
unsigned型定数	文字**u**または**U**を末尾につける	**12800U**
long型定数	文字**l**または**L**を末尾につける	**0L**
long long型定数	文字**ll**または**LL**を末尾につける	**3456LL**

注：(1) long long型と接尾語ll LLはC99での追加機能
(2) 接尾語Uと他の接尾語を組み合わせて指定できる。例：UL ULL

007 浮動小数点定数

定数3

浮動小数点定数は小数部をもつ数値定数です。浮動小数点定数には、小数点形式と指数形式があります。浮動小数点定数の型はデフォルトでdouble型であり、接尾語Fがあるとfloat型に、接尾語Lがあるとlong double型になります。

浮動小数点定数の表現

区別	説明	例
小数点形式	ピリオドを使って表現する	**12.345**
指数形式	**e**または**E**を使って表現する	**3.456E2**(=345.6) **6.78e-3**(=0.00678)
float型定数	文字**f**または**F**を末尾につける	**12.34F**　**1.234E2f**
long double型定数	文字**l**または**L**を末尾につける	**100.23L**　**5.67e-21**

C99では浮動小数点数として次のような、特殊値の表現形式があります。複数の候補のどれを使うか、また"*n*文字列"の意味は処理系定義です。小文字で表示される場合もあります。printf系関数はこの形式の出力を行ないます。scanf系関数はこの形式の入力を受け付けます。strtod()関数でも使用できます。マクロ名としてはINFINITYとNANが、math.hで定義されています。

INF INFINITY　——無限大を表す
NAN NAN(*n*文字列)　——非数を表す(Not-a-Number)

note 16進浮動小数点定数

C99では浮動小数点定数を16進数で表現できるようになった。次のように記述する。

0Xと**P**は小文字でもよい。**P**の後ろには符号の**+**または**-**をつけることができる。接尾語として**float**型を示す**f**または**F**, **long double**型を示す**l**または**L**をつけることができる。

0Xaaaa.bbbbPxx　——整数部**aaaa**, 小数部**bbbb**, 指数部**xx**(2の**xx**乗)の数を示す

プログラム例：16進浮動小数点定数と%a変換を使用する

```
#include <stdio.h>
#include <float.h>
int main(void)
{
    double a = 0x123.DEFp3;
    printf("a=%a¥n", a);
    printf("DBL_MAX=%a¥n", DBL_MAX);
    return 0;
}
```

実行結果
C99対応処理系で実行
```
a=0x1.23defp+11
DBL_MAX=0x1.
fffffffffffffp+1023
```

008 文字定数

定数4

文字定数は該当する文字を一重引用符(')で囲むことで表現します。先頭にLをつけるとワイド文字になります。文字定数の値は、その文字のもつ文字コードです。

```
int ch;
ch = 'A';          ——文字定数

wchar_t wch;
wch = L'漢';        ——ワイド文字定数
```

009 エスケープ表記

定数5

通常の方法では表示できない文字コードを表現するために、**エスケープ表記**(escape sequence)が用意されています(JIS規格用語では逆斜線表記)。エスケープ表記は文字定数や文字列リテラルの一部として用いることができます。

エスケープ表記

文字列	コード	機能	
¥a	07	警報	alert
¥b	08	後退	back space
¥f	0C	書式送り	form feed
¥n	0A	改行	new line
¥r	0D	復帰	carriage return
¥t	09	水平タブ	horizontal tab
¥v	0B	垂直タブ	vertical tab

文字列	コード	機能	
¥¥	5C	¥自身	
¥'	2C	'(文字定数対策)	
¥"	22	"(文字列リテラル対策)	
¥?	3F	?(3文字表記対策)	
¥ooo		8進数	octal
¥xhh		16進数	hexadecimal

注1：8進数表記は非8進文字に出合うと終了(最大3文字まで)。

注2：16進数表記は非16進文字に出合うと終了(長さに制限はない)。
"¥xabcd"で16進表記が¥xabであるときは"¥xab" "cd"と分離するとよい。

注3：文字列の終端文字として¥0という表現がよく使われる。
記述例：

```
ch = '¥n';       ——chは0Aになる
ch = '¥'';       ——chは2Cになる
ch = 'B';        ——chは42になる(文字Bのコード)
ch = '¥x42';     ——同上(文字Bの16進数表現)
ch = '¥102';     ——同上(文字Bの8進数表現)
ch = '¥0';       ——ch=0;と同じだが「文字コードの0」という意味が明示的
```

010 文字列リテラル

定数6

文字列リテラルは該当する文字列を二重引用符(")で囲むことで表現します。先頭にLをつけるとワイド文字列になります。隣接する文字列リテラルは連結され、ひとつの文字列とみなされます。

```
char s[80] = "文字列";              ――文字列リテラル
wchar_t s[80] = L"wide文字列";      ――ワイド文字列リテラル
char ss[] = "AB"  "12" "ab";        ――ssは"AB12ab"
```

文字列リテラルの値は、該当する文字コードの集合に終端を示す文字コード'¥0'を付加したものです。たとえば"ABC"の内部表現は次のようになります。

note 国際文字名の表現

C99では文字定数や文字列リテラルの中に**国際文字名**を表現する方法が用意された。8桁固定または4桁固定で、16進数文字を使って次のように記述する。大文字(**¥U**)と小文字(**¥u**)の使い分けをするので注意。

```
8桁固定の16進数表示：¥Uxxxxxxxx   ――コードxxxxxxxxである国際文字を示す
4桁固定の16進数表示：¥uxxxx       ――コード¥U0000xxxxである国際文字を示す
```

011 記号定数とconst定数

定数7

#defineを使うと定数に名前をつけて、記号定数として用いることができます。これはプログラムを明示的にするのに有益です。またconst修飾子を使うと、値を変更できない変数を指定できます。→「116 マクロ定義 #define」(p. 166),「const修飾子」(p. 30)

```
#define MAXSIZE 64000        ――64000にMAXSIZEという名前をつける
if (ct < MAXSIZE) ...        ――定義した記号定数を利用できる

const int a = 1234;          ――const修飾つき変数
a = 100;                     ――エラー。aの値は変更できない
```

012 複合リテラル

定数8

C99では**複合リテラル**機能が用意されています。複合リテラルは複数の初期化子で構成されるオブジェクトを名前なしで表現するものです。具体的には、

――かっこで囲まれた型名とそれに続く波かっこで囲まれた初期化子の並びで構成される後置式

のことをいいます。たとえば次のふたつの配列記述は同じように処理されます。

記述1

```
int d[] = {1, 2, 3, 4};          // 配列を用意し
func(d);                         // それを関数に渡す
```

記述2

```
func((int [])(1, 2, 3, 4});    // 複合リテラルで渡す
```

次の例は配列を直接ポインタで管理しています。

```
double *p = (double [])(11.22, 33.44, 55.66};
```

構造体の場合も複合リテラルを使って、次のように記述できます。

```
struct myst { int x, y; };       // 構造体
struct myst dt = { 1, 2 };       // myst型変数

func1(dt);                       // 通常の変数渡し
func1((struct myst){3, 4});      // 複合リテラルで渡す

func2(&(struct myst){5, 6});     // 複合リテラルのアドレスを渡す

return (struct myst){7, 8};      // 複合リテラルを戻り値にする
```

第3章

変数とデータ型

C Quick Reference

013 変数宣言

変数とデータ型1

Cでは**変数**は宣言してから使用します。変数宣言の例を示します。

変数宣言の例

```
int a;                              ——(1)
int b, c;                           ——(2)
static const unsigned short int d;  ——(3)
const int e;                        ——(4)
int const e;                        ——(5)
```

ここで(1)(2)は基本的な用法です。(3)は型を修飾するさまざまな指定子を用いた例です。指定子の順序は自由ですから、(4)は(5)のようにもできます。しかし(4)の方が、スタイルとしては自然です。

用意されている指定子には次のものがあります。typedefは特殊機能で変数宣言には用いません。また関数指定子は関数宣言のときだけ使用できます。inlineについては「095 インライン関数」(p. 135)で説明します。

宣言用指定子

記憶クラス指定子	auto static extern register typedef
型修飾子	const volatile restrict
型指定子	void char short int long float double signed unsigned _Bool _Complex _Imaginaly 構造体共用体指定子　列挙型指定子　型定義名
関数指定子	inline

注：restrict _Bool _Complex _Imaginaly inlineはC99の追加機能。

指定子の順序は自由ですが、例(3)で示したように、

[記憶クラス] [型修飾] [符号修飾] [サイズ修飾] 型名 識別子

にすると読みやすいようです。なお型名によっては使えない指定子組み合わせもあります。たとえば「unsigned double」とすることはできません。

note 不完全型とフレキシブル配列メンバ

C99規格では「型」は次の3つに分類される。

オブジェクト型　——オブジェクトを完全に規定する型
不完全型　——オブジェクトを規定する型で、その大きさを確定するのに必要な情報が欠けたもの
関数型　——関数を規定する型

大きさの分からない配列は不完全であり、これを**不完全配列型**という。この型はサイズをともなう宣言が、あとで（別の場所で）行なわれたときに「完全」となる。また内容の分からない構造体型も不完全型である。これもあとで内容を定義することで完全型になる。次に不完全配列型の例を示す。

```
extern int dt[];
```
——どこか別の場所で定義されるサイズ指定のない不完全配列型

構造体は、基本的には宣言をするときそのサイズが確定していなければならない。しかしふたつ以上の名前つきメンバをもつ構造体の最後のメンバは、不完全配列型をもってもよい。

これを**フレキシブル配列メンバ**という。→「105 フレキシブル配列メンバ」（p. 150）

014 変数宣言の位置

変数とデータ型2

Cでは変数は次の場所で宣言できます(C99以前)。

(1)関数の外
(2)関数内の先頭部分(文より前)
(3)関数内の任意のブロック{ }内の先頭部分(文より前)

(1)はグローバル変数を宣言するときに使います。(2)はローカル変数を宣言するときに使います。(3)はブロック{}内で宣言します。標準的な宣言は(1)(2)で行ない、必要に応じて(3)を使います。

次に各タイプの変数宣言例を示します。

```
#include <stdio.h>
int g;                ――(1)関数の外で宣言したグローバル変数
int main(void)
{
    int n;            ――(2)関数の中の先頭部分で宣言したローカル変数
    if (n > 0) {
        int x;        ――(3)任意のブロック{}の内部で宣言したローカル変数
        ...              変数xはこのブロック内のみで有効
    }
    ...
}
```

C99では次の機能が追加されました。

(a)上記の(2)(3)における「先頭部分」の制約がなくなり、任意位置での宣言が可能
(b)for文の式記述の中で宣言が可能(C99の追加機能)

C99対応の処理系での記述例を示します。

```
int a;                           ――変数宣言
a = 10;                          ――文の記述
int b;                           ――(a)再び変数宣言ができる
b = 20;                          ――文の記述

for (int n=1; n<=10; n++) {      ――(b)変数nを宣言。このnはforブロック内で有効
    ...
}
```

015 データ型

変数とデータ型3

符号修飾、サイズ修飾、および型名の、可能な組み合わせによるデータ型一覧を次に示します。ビット幅は一例であり、処理系により異なることがあります。実際にデータ型が何バイトで構成されているかはsizeof演算子を使って知ることができます。

```
printf("%d¥n", sizeof(int));          // int型のバイト幅を表示
printf("%d¥n", sizeof(long int));     // long int型のバイト幅を表示
```

データ型一覧

データ型	ビット幅の例	表現範囲の例	
char	8	-128～127	
signed char	8	-128～127	
unsigned char	8	0～255	
short int	16	-32768～32767	
signed short int	16	-32768～32767	
unsigned short int	16	0～65535	
int	32	-2147483648～2147483647	
signed int	32	-2147483648～2147483647	
unsigned int	32	0～4294967295	
long int	32	-2147483648～2147483647	
signed long int	32	-2147483648～2147483647	
unsigned long int	32	0～4294967295	
long long int	64	-9223372036854775808～9223372036854775807	
signed long long int	64	-9223372036854775808～9223372036854775807	
unsigned long long int	64	0～18446744073709551615	
float	32	およそ10の-38乗～10の38乗	有効6桁
double	64	およそ10の-308乗～10の308乗	有効15桁
long double	96	およそ10の-4932乗～10の4932乗	有効18桁

注1：char型の符号有無は実装依存だが「符号あり」の処理系が多い。

注2：long double型は実際には「double型と同じ64」の処理系もある。

注3：整数型のサイズは実装依存であるが、次のサイズ以上であることが規定されている。
　char(8)　short(16)　int(16)　long(32)　long long(64)

注4：複数の指定子で構成される整数型で、末尾がintの場合は、intを省略できる。
　例：shortはshort intと同じ。longはlong intと同じ

注5：long long int型はC99で追加になった。

016 拡張整数型

変数とデータ型4

整数型のうち、signed char, short int, int, long int, long long intの5種類を標準符号つき整数型といいます。これに対応してunsignedをもつ型を標準符号なし整数型といいます。両者を合わせて**標準整数型**といいます。

このようなC規格として備わっている標準型ではなく、処理系定義の型もあります。それらは符号があれば拡張符号つき整数型、なければ拡張符号なし整数型、両者を合わせて**拡張整数型**といいます。たとえばVisual C++では__int8, __int16, __int32などの拡張整数型があります。

またC99では特別の用途のための拡張整数型が用意されています。stdint.hとinttypes.hの各ヘッダファイルでサポートされており、そこでは、たとえばint_fast32_t整数型などが用意されています。→「L17 stdint.h 整数型処理」(p. 265)

通常のCプログラミングでは、これらの拡張整数型を使用することはあまりありません。

017 void型

変数とデータ型5

void型の指定

void型（void：空の）は、「値をもたないことを明示する」という特殊な目的で用いられます。また戻り値無視を表明する用法もあります。

```
void boo(int n)    ――戻り値がない関数
{
    ...
}

int foo(void)      ――引数がない関数
{
    ...
}

(void)func1();     ――戻り値を無視することを明示的にする
```

void型ポインタ

注：ポインタについては「第10章 ポインタ」(p. 97)で詳しく説明していますので、必要に応じてそちらを先に読んでください。

void型のポインタを宣言することができます。void型ポインタはどのような型のポインタも保持できます。void型ポインタを関数仮引数に使うこともできます(「092 データを渡す方法1」(p. 128))。

```
void *p;                    ──voidポインタpを宣言。どの型のポインタも保持できる
int foo(int n, void *p);    ──仮引数をvoid*型にする
```

void型のポインタは、
──とりあえずポインタ宣言しておいてデータ型はあとで決める
という使い方をします。

たとえば標準関数malloc()の書式は「void *malloc(size_t size);」であり、これはどのような型のメモリ確保に使用してもよいという意図をもっています。

void型ポインタにはどの型のポインタでも、そのまま代入できます。また逆方向のポインタ代入もできます。

```
void *vp;
char *cp;
int  *ip;

vp = cp;            ──正しい
vp = ip;            ──正しい
cp = vp;            ──正しい
ip = vp;            ──正しい
cp = ip;            ──正しいが警告されるかもしれない
cp = (char *)ip;    ──正しい。キャスト変換(後述)をすれば警告はない
```

018 列挙型データ

変数とデータ型6

列挙型の宣言

構文

```
enum 列挙体タグopt {列挙定数並び}
```

注：optのついた項目は省略可能であることを示す。以降も同様である。

列挙定数と呼ばれる整数値を上のルールで列挙したものを、**列挙体**といいます。その列挙体内容をもつ型を**列挙型**といいます。定義した列挙定数は、プログラムの中で整数値を記述可能な場所なら、どこでも用いることができます。

列挙定数の値は、デフォルトでは、先頭が0で、以降は、直前の列挙定数の値に+1したものです。また「xxx=値」の形で、値を指定することもできます。次に列挙型の記述例を示します。

```
enum color {red, blue, green, white};     // 列挙型の宣言
enum color cd;                            // color型の変数
int  id;                                  // int型の変数

cd = green;                               // color型変数にgreenを設定
if (cd == green) puts("YES");             // 出力：YES
id = white;                               // int型変数にwhiteを設定
if (id == white) puts("YES");             // 出力：YES
printf("%d %d¥n", cd, id);                // 出力：2 3
```

ここではint型変数に列挙定数を設定する例も示しています。もし列挙定数であることを明示したいときはidt=(enum color)white;としてもかまいません。

またC99規格では、最後の列挙定数の後ろに','をおいてもかまいません。これは便利のために用意されました。

```
enum color {red, blue, green, white, };  // 末尾の,もOK
```

列挙定数値を指定する例を次に示します。列挙定数値は重複してもかまいません。例2はそれを示しています。

例1

```
enum etype {aa, bb, cc=5, dd, ee};
printf("%d %d %d %d %d¥n", aa, bb, cc, dd, ee);     // 出力：0 1 5 6 7
```

例2

```
enum coltyp {red, blue, aka=0, ao};
printf("%d %d %d %d¥n", red, blue, aka, ao);    // 出力：0 1 0 1
```

列挙型のいろいろな宣言方法

列挙型はいろいろな宣言方法があります。次に例を示します。

例1：基本的な方法

```
enum color {red, blue, green};   // 型の宣言
enum color dt;                   // 変数の宣言
```

例2：列挙型宣言と同時に変数を宣言

```
enum color {red, blue, green} dt;
```

例3：複数の変数を宣言

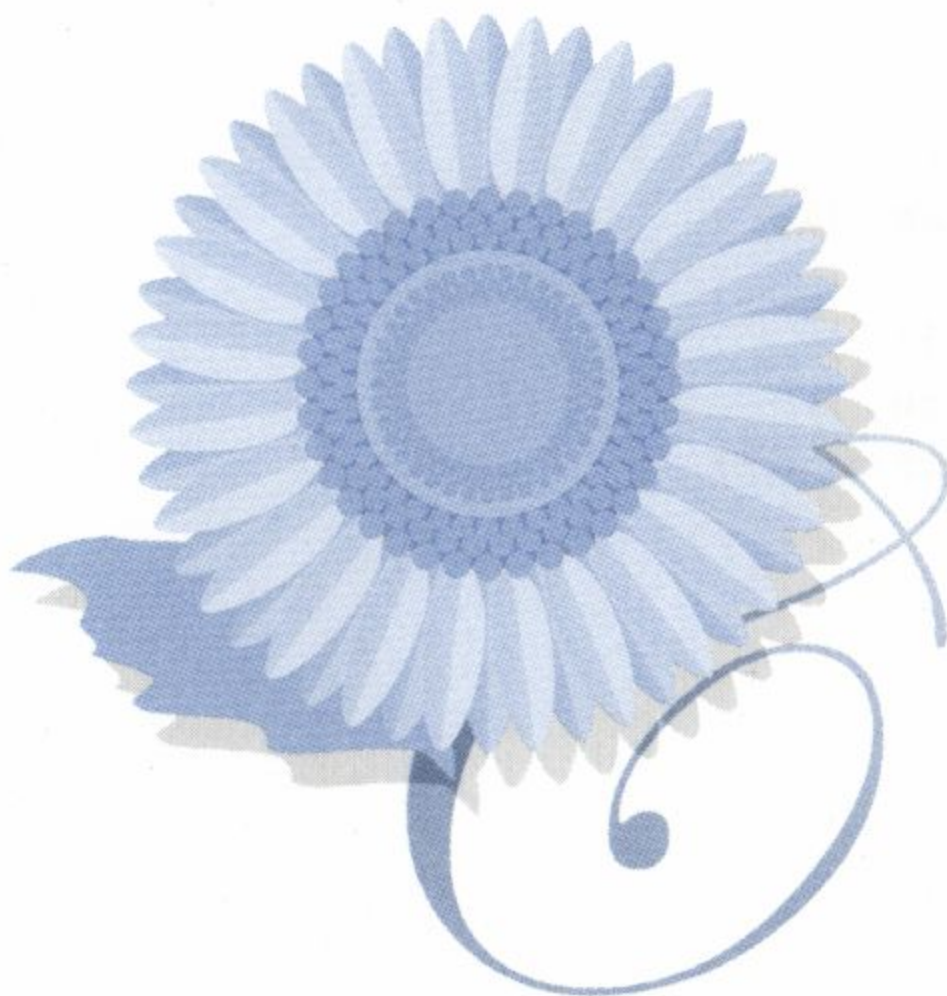
```
enum color {red, blue, green} d1, d2;
```

例4：列挙体名は指定せず、変数cdtを宣言。列挙型名をこのあと使用しない場合に使う

```
enum {red, blue, green} cdt;
    └──名前がない
```

例5：typedef宣言を使って変数宣言時のenum記述を不要にする

注：typedef機能については「021 typedef宣言」(p. 32) で説明する

```
typedef enum {red, blue, green} Color;    // typedefを使う
Color dt;                                 // enumが不要になる
```

019 C99で追加された型

変数とデータ型7

_Bool型

_Bool型は論理型の値を表現します。

```
_Bool d1;  // 論理型の変数d1を宣言。0/1の値を保持する
```

関連機能としてstdbool.hヘッダで次の4つのマクロが定義されています。既存Cコードとの識別子衝突を防止するために、ヘッダを指定したときだけ有効にしています。

```
bool                            → _Bool に展開
true                            → 整数定数 1 に展開
false                           → 整数定数 0 に展開
__bool_true_false_are_defined   → 整数定数 1 に展開
```

long long int型

long long intは-9223372036854775808～9223372036854775807の値を表現する整数型です。unsigned long long intもあります。

```
long long int d1;    // 変数d1を宣言
```

_Complex型と_Imaginary型

_Complex型は複素数を表現します。_Imaginary型は虚数部の値を表現(実数部が0の複素数型)します。次のように基本型と組み合わせて宣言します。

```
float _Complex d2;
double _Complex d3;
long double _Complex d4;

float _Imaginary d5;
double _Imaginary d6;
long double _Imaginary d7;
```

次のふたつのプログラムはC99をサポートしている処理系で実行できます。

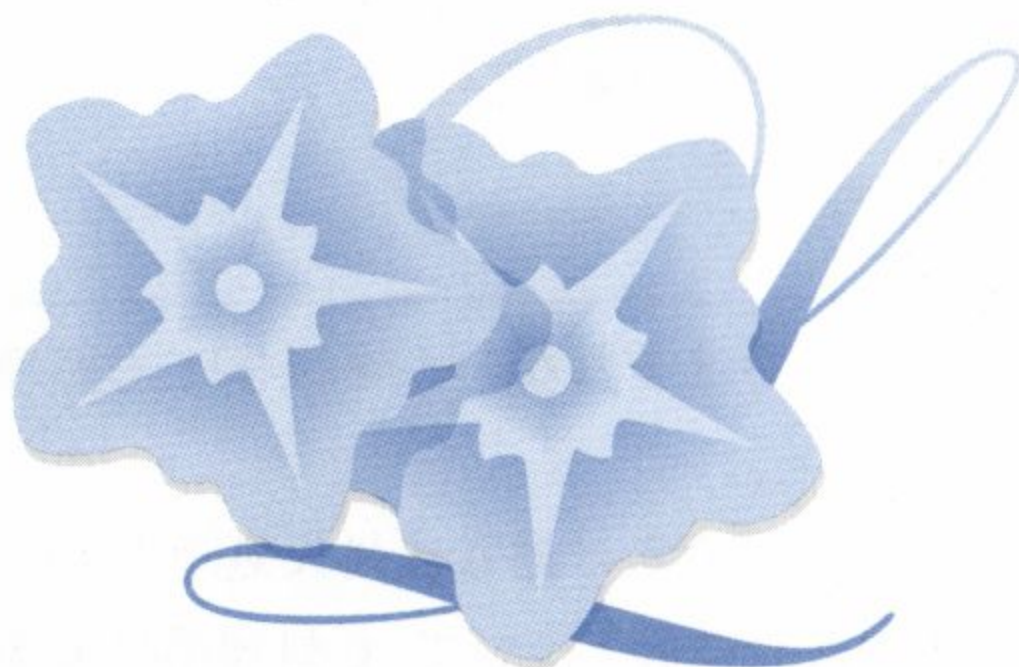
プログラム例：_Bool, true, long long intと接尾語LLを使う

```
#include <stdio.h>
#include <stdbool.h>
int main(void)
{
    _Bool f;
    long long int lldt;

    f = true;
    if (f) puts("yes"); else puts("no"); // 出力：yes
    f = !f;
    if (f) puts("yes"); else puts("no"); // 出力：no

    lldt = 85016406342000LL;
    printf("lldt=%lld¥n", lldt);          // 出力：lldt=85016406342000
    return 0;
}
```

プログラム例：_Complexを使う

```
#include <stdio.h>
#include <complex.h>
int main(void)
{
    double _Complex a = 1.0+2.0*I, b = 1.0-2.0*I;
                         // Iは虚数単位を表す定数
    a = a * b;           // creal()は実部、cimag()は虚部を返す関数
    printf("(1+2i)*(1-2i) = %f+%fi¥n", creal(a), cimag(a));
                         // 出力：(1+2i)*(1-2i) = 5.000000+0.000000i
    return 0;
}
```

020 型修飾子

変数とデータ型8

const修飾子

■変数のconst宣言

const修飾子はオブジェクトの値を変更できないことを指定します。次の例で、double型変数piの値は初期化で設定され、以降はプログラム的には変更できなくなります。

```
const double pi = 3.14159;      ――初期化で値設定
pi = 0.0;                       ――以降の変更はエラーになる
```

■仮引数のconst宣言

関数引数にconst修飾子を用いると、その引数は関数内で変更不可になります。const型の引数には、関数呼び出しのときに初期値が設定されます。またconstつき引数は、その内容が変更されないことを表明する効果もあります。次の関数原型は、複写方向が、「s1 ← s2」であることを暗示しています。

```
char *strcpy(char *s1, const char *s2);    ――s2は変更されない
```

■ポインタのconst宣言

ポインタに関連するconst宣言の場合、ポインタをconstにする方法と、ポインタで参照するオブジェクトをconstにする方法があります。次に例を示します。

```
int a=10, b=10;
int * const p1 = &a;      // p1がconst
const int *p2 = &b;       // 「*p2」がconst
                          // 修飾子の順番は自由なのでint const *p2でもよい

*p1 = 100;                // 値の設定は正しい
// p1 = &a;               // ポインタを変更するのでエラー
p2 = &a;                  // アドレスの設定は正しい
// *p2 = 100;             // p2の指す値である*p2を変更するのでエラー
```

volatile修飾子

volatile修飾子は(volatile：きまぐれな、一時的な)、C処理系に、

――最適化をしないでほしい

と伝える役目をもっています。変数は通常は代入処理や++, --演算が行なわれたときに値を変更します。この前提にもとづいて、C処理系は最適化処理を行ないます。

しかしハードウェアデバイスによる割込み処理などの、非同期プロセスで、特定のメモリ内容が書き換えられるときは、最適化が不適切になります。そのような場合、volatile修飾を行なうと、そのオブジェクトに関する最適化処理を抑止できます。

```
volatile int n;                        ——変数nに関する最適化は行なわない
volatile int *memdt = 適切な番地;      ——*memdtに関する最適化は行なわない
```

次のコードはvolatile修飾子の効果を見る簡単な例です。

Cコード

```
volatile int dt = 0;
int a, b;
...
a = dt;
b = dt;
```

volatileがある場合のアセンブラコード例

```
mov     eax, DWORD PTR _dt
mov     ecx, DWORD PTR _dt
mov     DWORD PTR _a, eax
mov     DWORD PTR _b, ecx
```

volatileがない場合のアセンブラコード例

```
mov     eax, DWORD PTR _dt    // 1回だけeaxにコピーしている
mov     DWORD PTR _a, eax
mov     DWORD PTR _b, eax
```

restrict修飾子

restrict修飾子は C99 で追加された機能で、複数のポインタが同一のオブジェクトを指していないことを明示します。

この機能は処理系に最適化のヒントを与える役目をもつもので、プログラマ側から見た場合、restrictの有無でプログラムの見た目の動作は変わりません。

たとえば標準関数であるmemcpy()は、ふたつの引数の指す領域が重ならないことを想定しています(重なりあうときの動作は未定義)。一方、memmove()関数は、ふたつの引数の指す領域が重なっても正しくコピー処理することを保証します。この両関数のプロトタイプは次のようになっています。

```
void *memcpy(void * restrict s1, const void * restrict s2, size_t n);
void *memmove(void *s1, const void *s2, size_t n);
```

021 typedef宣言

変数とデータ型9

typedef宣言は新しいデータ型の名前を作成する、つまり**型定義**(typeのdefinition)を行なうものです。新しい型そのものを作るのではなく、指定した型名の同義語を導入するものです。次に基本的な記述例を示します。

```
typedef int word;               ——int と同じ意味をもつ word を宣言
typedef unsigned int uint;      ——unsigned int と同じ意味をもつ uint を宣言

word a;                         ——int a; と同じ
uint b;                         ——unsigned int b; と同じ
```

typedef宣言は非常に柔軟な利用ができます。いろいろなパターンの利用例を示します。

例1:複数の同義語を宣言

```
typedef short int SINT, INT16;
SINT a;              ——short int a; と同じ
INT16 b;             ——short int b; と同じ
```

例2:ポインタの同義語を宣言

```
typedef int *intptr;
intptr p1, p2;       ——int *p, *p2; と同じ
```

例3:文字配列の同義語を宣言

```
typedef char String[];
String s1 = "abcde";     ——char s1[] = "abcde"; と同じ

typedef char Str256[256];
Str256 s2;               ——char s2[256]; と同じ
```

例4:二次元配列の同義語を宣言

```
typedef int INT55[5][5];
INT55 dt;                ——int dt[5][5]; と同じ
```

例5:「戻り値がvoid型でint型引数をもつ関数へのポインタ」の同義語を宣言

```
typedef void (*vi_fncP)(int);
vi_fncP myfnc;           ——void (*myfnc)(int); と同じ
```

例6:構造体用の同義語を宣言

```
struct styp { int a, b; };
typedef struct styp Styp;
Styp dt;                 ——struct styp dt; と同じ
```

例7:構造体宣言と同時に記述

```
typedef struct styp { int a, b; } Styp;
Styp dt;                 ——struct styp dt; と同じ
```

第4章

配列と文字列

C Quick Reference

022 一次元配列

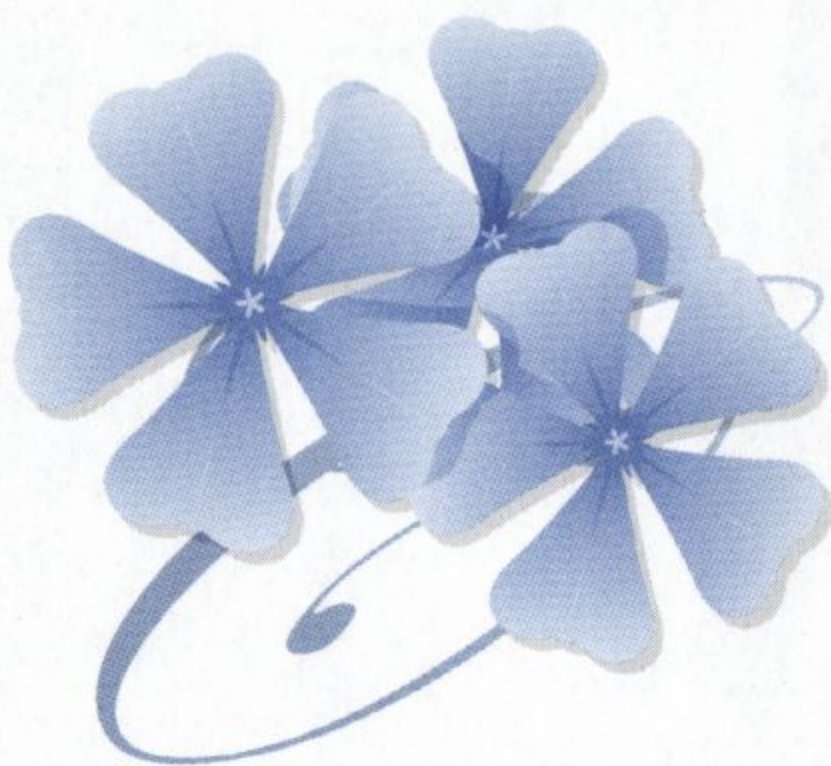
配列と文字列 1

配列は同じ型のデータを集めて名前をつけたものです。配列を宣言するときは、配列長を指定します。

```
int a[10];              ――配列長10のint型配列
int b[20], c[30];       ――配列長20のint型配列と配列長30のint型配列
char s[80];             ――配列長80のchar型配列
```

配列の添字は0からはじまります。Cコンパイラは実行時に配列の境界をチェックしません。境界を越えた操作をしないように注意する必要があります。

```
int n[10];              ――n[0]～n[9]が有効
n[0] = 123;             ――先頭要素に値設定
n[9] = 789;             ――末尾要素に値設定
n[15] = 99;             ――不正だがコンパイルエラーにはならないので注意
```

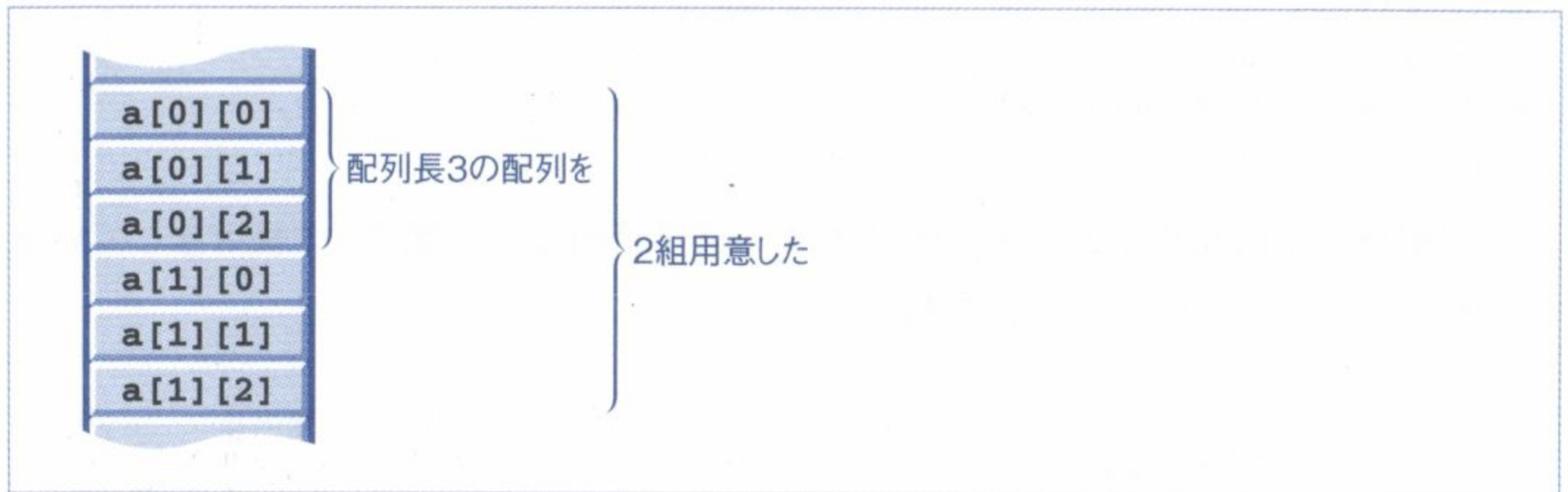
023 多次元配列

配列は多次元にすることもできます。次元数は [] をいくつも連ねることで指定します。

```
int a[2][3];             ——(1) int型の二次元配列
double b[5][8][10];      ——(2) double型の三次元配列
```

ここで(1)の指定は、「配列長3の配列を2組用意」という確保です。多次元配列はメモリ上に、「後ろの添字を先に変化させる」というルールで配置されます。

int a[2][3];のメモリ配置

024 文字配列

配列と文字列3

文字列の格納にはchar型配列を利用します。たとえばASCII文字を80文字分だけ格納したいときは、次のようにします。

```
char s[80];
```

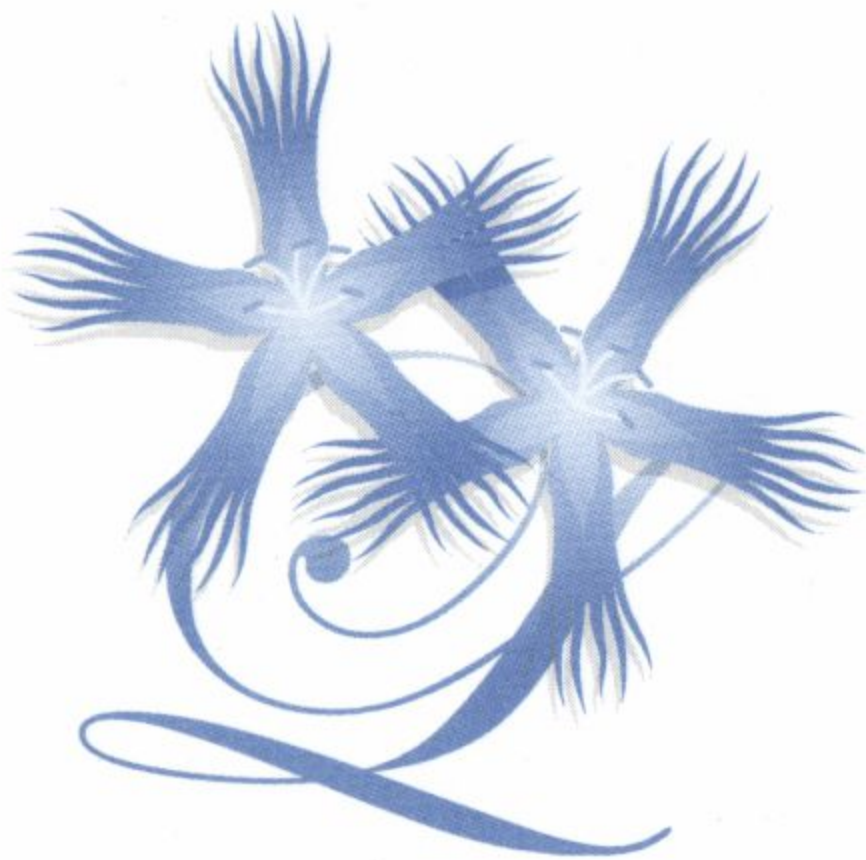
この配列には79文字が格納可能です。C言語の約束により文字列の最後に**ナル文字**（またはヌル文字）'¥0'が付加されるからです。文字列用の配列は、必要な文字数よりも、余裕をもった長さを確保する方が安全です。格納可能な文字数を正しく把握しておくのはプログラマの責任です。

```
char s[10];                         ——配列長が10
strcpy(s, "abcde");                 ——正しい文字列設定
strcpy(s, "abcdefghijklm");         ——不正。配列あふれになる
```

文字配列も多次元化できます。次の例は80文字格納できる配列を4組用意しています。それぞれss[0]〜ss[3]で指定します。

```
char ss[4][80];                     ——80文字まで格納できる配列を4組用意
strcpy(ss[0], "aaaa");              ——最初の配列要素に文字列設定
puts(ss[0]);                        ——出力は"aaaa"
```

025 可変長配列

配列と文字列4

C99では変数でサイズ指定をする動的な配列確保が可能になりました。これを**可変長配列**といいます。可変長配列は、実行時に配列サイズが決定され、そのサイズはオブジェクトの生存期間の間変わることはありません。次に可変長配列の記述例を示します。

```
void func1(int n)
{
    char a[n];        // サイズnであるchar型可変長配列
    int  b[n+10];     // サイズn+10であるint型可変長配列
    ...
}
```

また関数の仮引数が可変長配列である（可変長配列仮引数）ことを示すときは、関数プロトタイプを [*] で記述することもできます。

可変長配列仮引数をもつ関数の例を示します。

可変長配列引数を使う関数

```
#include <stdio.h>
void foo(int n1, int n2, int ary[n1][n2]);   // 関数原型
//void foo(int, int, int [*][*]);            // これでもよい

void foo(int n1, int n2, int ary[n1][n2])
{
    int i,j;
    for(i=0; i<n1; i++) {
        for(j=0; j<n2; j++) {
            printf("%2d ", ary[i][j]);
        }
        printf("¥n");
    }
}

int main(void)
{
    int dt23[2][3] = { {11, 12, 13}, {21, 22, 23} };
    int dt32[3][2] = { {11, 12}, {21, 22}, {31, 32} };
    foo(2, 3, dt23);
    foo(3, 2, dt32);
    return 0;
}
```

第5章

型変換

C Quick Reference

026 暗黙の型変換

型変換 1

Cでは異なる型の間でのデータのやりとりや演算ができます。この場合、数値は暗黙のうちに、あるいは明示的に**型変換**が行なわれます。

まず**暗黙の型変換**について説明します。これはプログラマが特に意識しなくても、コンパイラが自動的に無難な型変換をしてくれるものです。この自動変換は代入と演算のふたつのケースで発生します。

代入時型変換

代入処理をすると右辺の値は左辺のデータ型に変換されます。

```
char   chr_dt;
int    int_dt;
float  flt_dt;
double dbl_dt;

int_dt = chr_dt; ――(1)
dbl_dt = flt_dt; ――(2)
dbl_dt = int_dt; ――(3)
chr_dt = int_dt; ――(4)
flt_dt = dbl_dt; ――(5)
int_dt = dbl_dt; ――(6)
```

(1)(2)(3)は「左辺の方が右辺より表現能力が高い」ので変換上の問題はありません。一方、(3)(4)(5)の場合は表現能力の低い型への変換」なので、

- 上位ビットの切捨て
- 小数部の切捨て
- 丸め処理

といったやむをえぬ処置が発生することがあります。どのようにデータが失われるかは処理系定義です。

Cでは表現能力の低い型への変換が行なわれた場合もエラーにはしません。通常は警告レベルを強くしてコンパイルすると、たとえば次のような警告表示が行なわれます。

```
'double' から 'int' への変換です。データが失われる可能性があります。
```

関数の引数型変換

関数を呼び出すときにも必要なら型変換が発生します。このときは実引数が、関数の仮引数のデータ型に変換されます。実引数と仮引数間で「代入するのと同じように」変換処理されます。次の例では、整数123がdouble型に変換されて、仮引数dtの値になります。

```
func(123);
...
void func(double dt)
{
    ...
}
```

単項変換

整数型はいくつも種類があり、細分化されていると効率が悪いため、まず単項の状態で標準的な型に変換されます。これを**単項変換**といいます。実際には次のように**整数拡張**(整数昇格)が行なわれます。

整数拡張

(1)**int**型より表現能力の低い整数型(**char**, **short int**, 列挙型、ビットフィールドなど)は**int**に変換される
(2)もし**int**で表現できない場合は**unsigned int**に変換される

配列名は特殊な場合(sizeof演算子や&演算子)を除いて、確保配列の先頭アドレスを示すポインタとして評価されます。→「ポインタ表現と配列表現」(p. 104)

また関数名もポインタとして評価されます。「Tを返す関数」があるとき、その単項変換結果は「Tを返す関数へのポインタ」です。→「note 関数指示子と&演算子と*演算子」(p. 111)

算術型変換

単項変換された変数は次に**算術型変換(二項変換)**されます。これは二項演算のときに行なわれる変換です。算術型変換では被演算数を、表現能力の高い方の型に統一します。その統一された状態で演算が行なわれます。

浮動小数点型の場合は、表現能力の序列は明確ですが、整数型の場合は種類が増え(拡張整数型もある)、符号有無があるため、簡単ではありません。C99では整数型に「整数変換の順位」という概念を導入しました。C規格には変換順位付けのルールが示されており、順位が高い方が表現能力が高くなります。次に標準の整数型の変換順位を示します。

整数変換の順位：標準の整数型のみ

long long, unsigned long long
long, unsigned long
int, unsigned int
short, unsigned short
char, unsigned char, signed char
_Bool

順位付けのルール（抜粋）

- 表現能力の高い符号つき整数型を高い順位にする
- 符号あり整数型と対応する符号なし整数型は同じ順位にする
- 標準整数型は同じ幅の拡張整数型より高い順位にする
- _Bool型はすべての標準整数型より低い順位にする

これらの順位と、符号有無、片方が他方のすべての値を表現可能か、によって変換方法を決定していきます。そのルールの簡略表現を次に示します。

算術変換ルール：最初に適合した処理を行なう

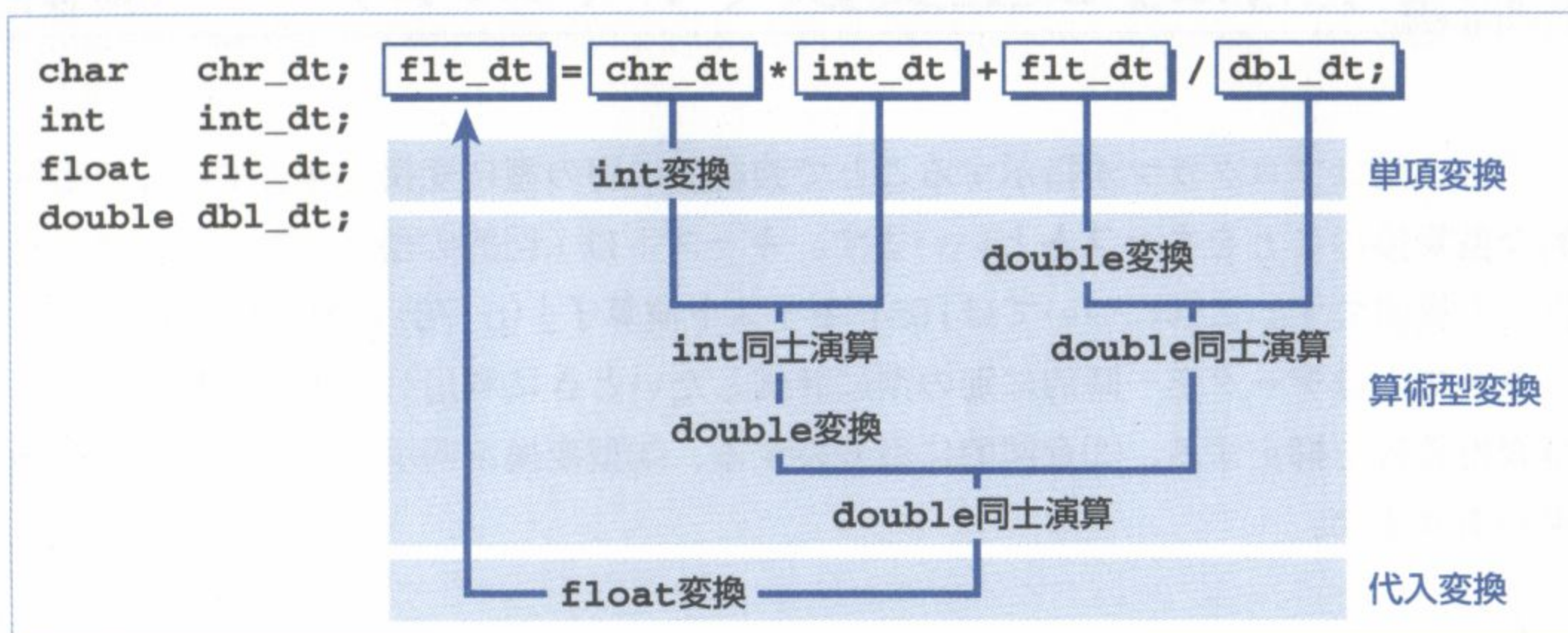
(1)両方が同じ型なら型変換は不要
(2)一方がlong doubleなら　他方をlong doubleに変換
(3)一方がdoubleなら　他方をdoubleに変換
(4)一方がfloatなら　他方をfloatに変換
(5)整数拡張を行ない、以下の変換処理を続行
(6)両方が同じ型なら型変換は不要
(7)両方が符号あり整数型、または両方が符号なし整数型なら、変換順位の高い方に変換
(8)符号なし整数型の変換順位>=他方の型の変換順位なら、他方を符号なし整数型に変換
(9)符号あり整数型が、他方の符号なし整数型のすべての値を表現できるなら、他方を符号あり整数型に変換
(10)一方の符号あり整数型に対応する符号なし整数型に両方を変換

このようにルールは複雑ですが、標準型の場合、実用的には、次の順序で表現可能な高い方の型に変換される、と考えればいいでしょう。

```
int → unsigned int → long → unsigned long → long long
    → unsigned long long → float → double → long double
```

型変換例：代入変換で警告は出るかもしれないが正しくコンパイルされる

027 明示的な型変換（キャスト）

型変換2

データ型はプログラマが指示することで強制的に別の型に変換できます。この**明示的な型変換**のことを**キャスト**といいます。キャストは（目的の型）のスタイルで型変換する機能です。これについては「054 キャスト演算子」（p. 78）に説明があります。

キャストはデータを一時的に別の型に評価したいときに利用します。これにより⑴型変換警告を抑止する、⑵意図的に型変換する、⑶型変換を明示的にする、などの効果があります。

```
char   cdt, *cp;
int    idt, *ip;
double ddt;

cdt = idt;            ――正しい（警告はされるかもしれない）
cdt = (char)idt;      ――正しい（警告なし）
ddt = (int)ddt;       ――キャストで整数部だけを取得する用法
myfnc((int)ddt);      ――myfnc()関数の仮引数がint型であることを明示している

ip = cp;              ――正しい（警告はされるかもしれない）
ip = (int *)cp;       ――正しい（警告なし）。明示的である
```

第6章

記憶クラス

C Quick Reference

028 識別子の有効範囲と可視性

記憶クラス1

オブジェクトや関数や構造体や型定義(typedef)などを表す名前である識別子は、それぞれ**有効範囲(スコープ)**をもっています。有効範囲は次の4つからなります(プリプロセッサ関連の識別子は除く)。

識別子の有効範囲

関数有効範囲	「ラベル名」が唯一該当する。ラベル名はそれが用いられる関数内のどの位置で(**goto**文とともに)参照されてもよい
ファイル有効範囲	識別子がブロック内になく仮引数でもないときはファイル有効範囲をもつ(たとえばグローバル変数名)。その宣言位置からファイルの終了位置(翻訳単位の終了位置)まで有効
ブロック有効範囲	関数の仮引数または、ブロックの内部で宣言された識別子は、その宣言位置からブロックの終了まで有効
関数原型有効範囲	関数原型(関数プロトタイプ)の仮引数として現れる識別子は、その宣言位置から関数原型の終了まで有効

注:翻訳単位については「032 識別子の結合と翻訳単位」(p. 49)を参照。

また同一名の識別子の有効範囲が重なる場合、内側の有効範囲にある名前が優先して可視になり、外側の有効範囲にある識別子は不可視(隠される)になります。

```
int a, b;           // (1) グローバル変数の宣言

void foo(void)
{
    double a;       // (2) ローカル変数宣言
    a = 12.34;      // これは(2)の変数a。ここで(1)の変数aは不可視
    b = 100;        // これは(1)の変数b
    if (b == 0) {
        char a;     // (3) 内部のブロックでの宣言
        a = 'C';    // これは(3)の変数a。ここで(1)と(2)の変数aは不可視
        ...
    }
    ...
}
```

宣言位置は文字位置単位で考えるので、同一行内でも前後の位置関係をもちます。次に例で示します。

```
int dt, *p = &dt;      ——正しい。dtが先に宣言されている
int *p = &dt, dt;      ——エラー。まだ宣言されていないdtを参照している
```

029 記憶クラスの種類

記憶クラス2

記憶クラスは変数の確保の方法、データ維持の方法、その通用範囲などを決めるものです。記憶クラスを決定する**記憶クラス指定子**には次のものがあります。

記憶クラス指定子

```
auto  static  extern  register  typedef
```

このうちtypedefは分類上、記憶クラスに入っていますが、構文的な都合によるものであり、記憶領域に関連した機能ではありません。typedefについては「021 typedef宣言」(p. 32)で説明しています。

030 識別子の宣言と定義

記憶クラス3

識別子(変数名など)の宣言は、厳密には純粋な宣言と、宣言&定義という意味をふくんでいます。宣言&定義のことを、通常は単に定義ということもあります。

宣言

識別子の解釈や属性を指定する

定義

宣言のうち実体をともなうものをいう。たとえば変数の場合はそのオブジェクトの記憶域を確保する宣言、関数の場合はその関数本体を含む宣言を、定義(宣言&定義)という

```
extern int a;        ――変数の宣言
int a=10;            ――変数の定義

void foo(void);      ――関数の宣言
void foo(void)       ――関数の定義
{
    ...
}
```

031 ローカル変数とグローバル変数 記憶クラス4

Cでは関数の内部で宣言した変数は、その関数の中だけで有効です。このように局所で通用する変数を**ローカル変数**(局所変数)といいます。一方、関数の外で宣言された変数は、同じファイル内のどの関数からも使用することができます。このような広域で通用する変数を**グローバル変数**(広域変数)といいます。

ローカル変数とグローバル変数

ローカル変数	関数内で宣言され、その関数内のみで参照できる
グローバル変数	関数外で宣言され、(翻訳単位内の)どの関数からも参照できる

グローバル変数とローカル変数の名前が重複しているときは、ローカル変数が優先されます。この関係を示します。

```
int g = 10;                          // グローバル変数g

void foo1(void)
{
    int g = 20, n = 30;              // ローカル変数gとn
    printf("g=%d n=%d¥n", g, n);     // 出力：g=20 n=30
}

void foo2(void)
{
    int n = 40;                      // ローカル変数n
    printf("g=%d n=%d¥n", g, n);     // 出力：g=10 n=40
}
```

032 識別子の結合と翻訳単位　記憶クラス5

前項で説明したグローバル変数は「どの関数からでも参照できる」としましたが、これは複数のソースファイルで構成されるときは制約を受けることがあります。プログラムが複数のxxx.cファイルで構成され、Cコンパイラがそれぞれを別個にコンパイルしてからリンクし、最終的にひとつの実行可能ファイルを得るとします。この1回ずつのコンパイル対象を「翻訳単位」といいます。翻訳単位とは、

――前処理後のプログラムテキスト

を単位とするものです。たとえばmyprg.cというプログラムが、

```
#include <stdio.h>
#include "myhead.h"
#include "mycode.c"
int main(void)
{
    ...
}
```

という内容であったとき、前処理を行なうと、プリプロセッサの働きで、stdio.hとmyhead.hとmycode.cのファイル内容が取り込まれて、ひとつのプログラムテキストファイルができます。これが**翻訳単位**になります。

Cではこの翻訳単位によって変数の通用範囲が変化します。

プログラムが複数のファイルで構成されるとき、翻訳単位間で、相手の識別子が参照できるかどうかで、**外部結合**、**内部結合**、**無結合**という3種類の結合状態があります。例としてa.cとb.cというソースファイルがあるものとして説明します。

ファイルの結合状態

外部結合	**a.c**の中で宣言された識別子を**b.c**でも参照できるという結合をいう リンカに識別子情報を渡す。識別子の実体は全体でひとつしかない 例：**extern**のついたグローバル変数は外部結合である
内部結合	**a.c**の中で宣言された識別子を**b.c**では参照できないという結合をいう 両方で同じ名前の識別子を衝突なしに使うことができる 例：**static**のついたグローバル変数は内部結合である
無結合	複数ファイル間の結合とは無関係な状態をいう 例：関数内で宣言されるローカル変数や仮引数は無結合である

033 記憶域期間

記憶クラス6

オブジェクトは「有効である期間」をもちます。これを**記憶域期間**といいます。記憶域期間には次のものがあります。

記憶期間

静的記憶域期間	プログラム実行中に常に存在するという期間 例：グローバル変数、および**static**つきで宣言された変数は静的記憶域期間をもつ
自動記憶域期間	関数が駆動され、そのブロックが有効なときだけ存在するという期間 例：関数内で**static**なしで宣言された変数は自動記憶域期間をもつ
割付け記憶域期間	記憶域管理関数で確保された記憶域が、解放されるまで存在するという期間 例：**malloc()**で確保し、**free()**で解放するまでのオブジェクトは割付け記憶域期間をもつ

034 auto指定子、register指定子 記憶クラス7

auto記憶クラス指定子は関数内での変数宣言に使用できます。関数が駆動され、そのブロックが有効になるとオブジェクトが用意され、ブロックが無効になると自動的に解放されます。autoは自動記憶域期間を指定するもので、局所的という性格をもちます。autoである変数を自動変数ともいいます。

このautoは省略可能です。関数内で宣言されていて、記憶クラス指定子のないオブジェクトはautoとみなされます。通常は省略するのが普通です。

```
void foo(void)
{
    auto int a;     ──int型の自動変数a
    int b;          ──int型の自動変数b。"auto"を省略している
    double c;       ──double型の自動変数c
    ...
}
```

register記憶クラス指定子は、そもそもは「可能ならレジスタに割り当てる」ことを指定するものでした。現在では、「registerつきで宣言されたオブジェクトへのアクセスは可能な限り高速にすべき」というヒントをコンパイラに与える役目をもちます。

register指定はautoを記述できる位置で使用できます。C規格では、このregister指定のあつかいは処理系定義であり「autoと同等のものとして処理してよい」ことになっています。

最適化技術の進歩した現在のコンパイラは、必要に応じて適切にレジスタ割り当てを行なうので、あえてregister指定を行なうことはありません。

```
register int a;    ──レジスタ変数であるint型のaを指定
int a;             ──コンパイラはこのように解釈するかもしれない
```

なおレジスタ変数はレジスタ上に配置されるものなので、変数のアドレスを取得するなどの操作はできません。また構文的には、register指定子を関数仮引数に使用できます。

```
register int b;
int *p = &b;                  ──誤り。アドレス取得はできない
void foo(register int n);     ──OK。仮引数に使用できる
```

035 extern指定子

記憶クラス8

extern記憶クラス指定子は、識別子（変数名や関数名）を外部結合するように指定するものです。

ただしこのextern指定子はデフォルト処理されます。関数の場合はexternがデフォルトであり、通常はextern指定を省略します。

またグローバルなオブジェクト（変数）宣言の場合も、記憶域クラス指定子をもたないときは外部結合になります。オブジェクトの場合は、明示的にするために適切にexternを付与するのが分かりやすいと思います。

異なる翻訳単位間で同じ変数宣言をし、どちらもexternがない場合、片方に初期化記述があれば、そちらが変数定義（実体がある）となり、他方は宣言となります。どちらにも初期化がない場合、どちらか一方が「定義」になるよう調整されます。どちらにも初期化がある場合はエラーです。以下、記述例で説明します。prg1.cとprg2.cは翻訳単位であり、

①**prg1.c**をコンパイルする
②**prg2.c**をコンパイルする
③できたふたつのオブジェクトファイルをリンクする

という手順で実行可能ファイルになるものとします。変数gはグローバル位置に記述されているものとします。

例1：externの基本的な用法

```
prg1.c：int g;              ――定義（実体確保）
prg2.c：extern int g;       ――宣言（参照する）
```

例2：externを使わず初期値有無で区別する方法

```
prg1.c：int g = 10;         ――初期化した側が定義になる
prg2.c：int g;              ――宣言（参照する）
```

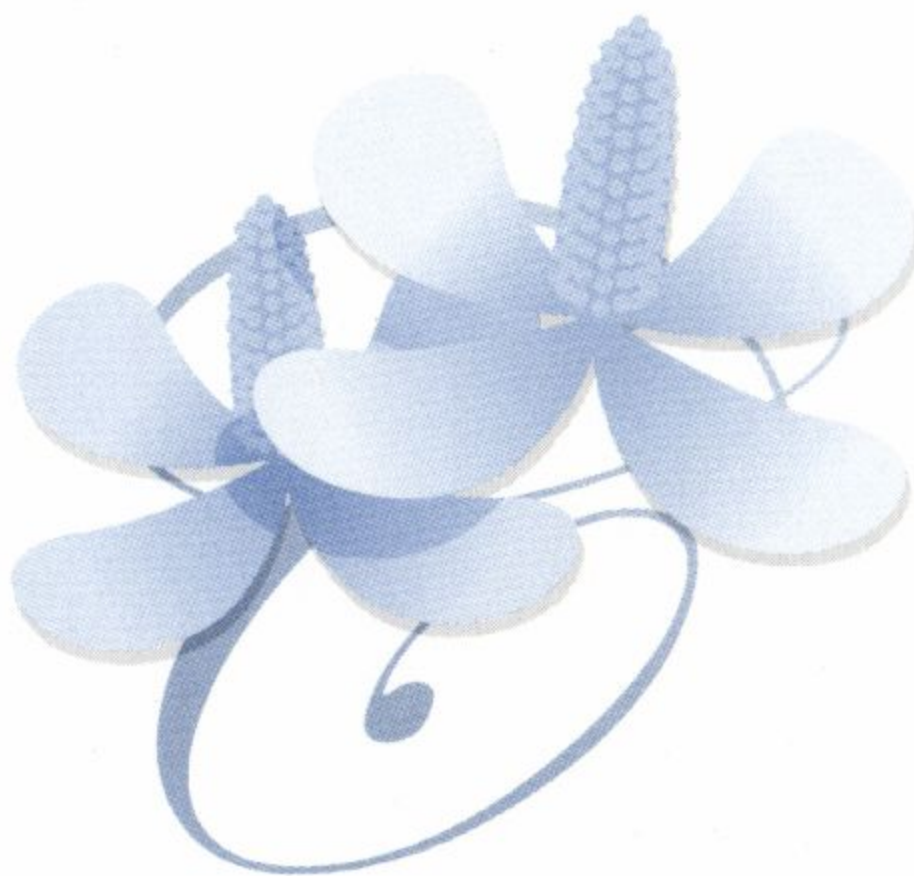
例3：externを使わずどちらも初期化がない

```
prg1.c：int g;   ｝片方が定義、他方が宣言に調整される
prg2.c：int g;   ｝
```

例4：ブロックの中でextern指定された変数は、そのブロック内のみで外部結合になる

```
void foo(void)
{
    extern int n;          ――このブロック内でのみ外部参照できる
    ...
}
```

例5：関数におけるextern指定

```
extern void foo(void) { ... }   ――外部結合の関数
void foo(void) { ... }          ――同上。externを省略しても同じ
```

036 static指定子

記憶クラス9

static記憶クラス指定子は、識別子（変数名や関数名）を内部結合するように指定するものです。また`static`のついた変数は静的記憶域期間をもちます。ローカル変数の場合も`static`がつけば静的記憶域期間をもちます。`static`のついたグローバルな名前は、

――自分の翻訳単位内においてはどの関数からも参照したいが、別の翻訳単位からは参照されたくない（別の翻訳単位とは名前の衝突を起こしたくない）

というときに用いられます。これはコンパイル処理としては、「この変数名はリンカに渡さないでくれ」といっていることになります。

以下、記述例で説明します。

例1：通常の関数記述

```
prg1.c：void foo(int n) { ... }; ――関数定義
prg2.c：foo(10);                 ――デフォルトで参照可能
```

例2：staticのついた関数

```
prg1.c：static void foo(void) { ... }; ｝両関数は別物である。それぞれの翻訳単位
prg2.c：static void foo(void) { ... }; ｝内で独自の関数となる。名前の衝突もない
```

例3：staticのついた変数

```
prg1.c：static int a;   ｝両変数は別物である。それぞれの翻訳単位内で
prg2.c：static int a;   ｝独自の変数となる。名前の衝突もない
```

例4：staticのついたローカル変数

```
void foo(void)
{
    static int a  = 10;     ――最初の駆動で10に設定。そのあとは前回の値を保持
    int b = 10;             ――自動変数なので駆動されるたびに10を設定
    ...
}
```

037 識別子の名前空間

記憶クラス10

識別子はここで説明したようにグローバルかローカルかで、通用範囲が異なりますが、これとは別にC言語には**名前空間**という概念もあります。これは識別子をグループ分けしたもので、名前空間が異なると、同じ識別子でも別のものとして区別されます。C言語の名前空間には、次の4つがあります。

なお、プリプロセッサのマクロ名は独自の名前空間になりますが、前処理後には次の4つの名前空間で管理されます。

C言語の名前空間

(1) ラベル名
(2) 構造体、共用体、列挙型のタグ
(3) 構造体や共用体のメンバ
(4) その他の一般識別子

名前空間を使い分けた例を示します。次のプログラムで識別子として4種のnameを使っていますが、これらはすべて名前空間が異なるので衝突しません。

名前空間で使い分けた識別子

```
#include <stdio.h>

struct name {  ――[タグ] の名前空間にある name
    int name;  ――[メンバ] の名前空間にある name
};

int main(void)
{                    ┌――[一般識別子] の名前空間にある name
    struct name name;

    name.name = 100;
    printf("name.name=%d¥n", name.name);
    goto name;
    printf("before name¥n");
name:                ――[ラベル名] の名前空間にある name
    printf("after name¥n");
    return 0;
}
```

実行結果

```
name.name=100
after name
```

第7章

初期化

C Quick Reference

038 初期化の方法

初期化1

初期化の形式

変数は宣言するときに、値を設定できます。これを**初期化**といいます。初期化の一般形は次のとおりです。

初期化の形式

```
宣言子 = 初期化子
```

初期化の例

```
int a = 100;  ——初期値を100にする
```

この例は単純変数の初期化です。配列や文字配列にはそれぞれ特別の初期化形式があります。

初期化のタイミング

初期化は次の2種類のタイミングで実行されます。

初期化のタイミング

自動変数とレジスタ変数	定義されている関数あるいはブロックに入るごとに初期化される
グローバル変数と静的変数	そのプログラムが実行開始されたときに一度だけ初期化される

暗黙の初期化

変数の初期化記述が行なわれていない場合には、次の基準で暗黙の初期化が行なわれます。以降、「自動変数」という表現はレジスタ変数もふくむものとします。

初期化記述をしないときの値

自動変数	→ 不定の値
グローバル変数と静的変数	→ 0系の値

自動変数はそれが必要となったとき、メモリ上に配置されます。そしてその初期値は配置されたメモリアドレス上にたまたま存在した値になります。静的変数とグローバル変数で初期値が指定されない場合は、自動的に0系の値が設定されます。つまり数値は0に、文字配列は""に、ポインタはNULLになります。たとえば次のとおりです。

```
int g;                    ——初期値は0
int *p;                   ——初期値はNULL
char s[80];               ——初期値は""
void foo(void)
{
    int a;                ——初期値は不定値
    static int b;         ——初期値は0
    ...
}
```

初期化に使う定数式と式

初期値は次のように、定数式(定数として評価できる式)、または式(定数式をふくむ任意の式)で指定します。静的記憶域期間をもつオブジェクト(グローバル変数や静的変数)は定数式または文字列リテラルで行ない、その初期値はコンパイル時に決定します。

初期値の指定

自動変数 → 式で指定
グローバル変数と静的変数 → 定数式や文字列リテラルで指定

初期値の指定の例

```
int a = 10;         ——定数式で初期化
int b = 10 + 20;    ——定数式で初期化
int c = a;          ——式で初期化
int d = a + 30;     ——式で初期化
```

039 単純変数の初期化

初期化2

配列や構造体などではない単純な変数のいろいろな初期化例を示します。

```
int g1 = 100;                    } グローバル変数の初期化例
int g2 = 'A' + 10;

void foo2(void)
{
    int a = 10;                  }
    int b = a + 20;              } 自動変数の初期化例
    int c = getchar();           }
    int r = rand();              }
    static int ct = -1;          ――静的変数の初期化例
    ...
}
```

note 初期化と{}の使用

後述するように配列や構造体などの初期化では **{}** を使用する。この **{}** は規格としてはスカラーオブジェクトや文字配列の初期化に用いてもよい（しかし、利用されることはあまりない）。

```
int a = 10;                      ――通常の記述
int a = { 10 };                  ――これでもよい

char ss[80] = "abcd";            ――通常の記述
char ss[80] = { "abcd" };        ――これでもよい

int n[3] = { 10 ,20, 30 };       ――配列の場合は {} が必須
```

040 配列の初期化

初期化3

一次元配列の初期化

配列は定数式からなる初期値リストを{}で囲むことで初期化できます。初期値リストはカンマで区切って書きます。自動配列、静的配列、グローバル変数の配列も初期化の方法は同じです。

ただしC99では自動記憶域期間をもつ配列を「式」で初期化できるようになりました。

一次元配列の初期化は次のスタイルで行ないます。

```
int a[4] = { 10, 20, 30, 40 };    ――(1)a[0]～a[3]に値を設定
int a[4] = { 10, 20, 30, 40, };   ――(2)最後のカンマがあってもよい
```

(2)で示すように、便宜のために最後の初期値の後ろに、カンマがあってもかまいません。これは、

```
int b[9] = { 10, 20, 30,
             40, 50, 60,
             70, 80, 90,          ――末尾のカンマが許される
           };
```

のように行分けして書くときに便利です。

サイズの省略と初期値省略

配列のサイズは省略できます。その場合は初期値の数から判断されます。

初期値リストはすべてを記述する必要はありません。初期値の数が要素数より少ないときは、不足する要素は0で初期化されます。初期値の数が要素数より多いときはエラーになります。

いろいろな配列初期化の例をあげます。

宣言	配列確保	初期値
int a[4] = { 1, 2, 3, 4 };	a[0]～a[3]	1, 2, 3, 4
int a[4] = { 1, 2, 3, 4, };	同上	同上
int b[] = { 11, 12, 13, 14 };	b[0]～b[3]	11, 12, 13, 14
int c[6] = { 21, 22, 23, 24 };	c[0]～c[5]	21, 22, 23, 24, 0, 0
int c[2] = { 21, 22, 23, 24 };	エラーになる	
double d[3] = { 1.2, 3.4, 5.6 };	d[0]～d[2]	1.2, 3.4, 5.6
char e[] = { 'A', 'B', 'C' };	e[0]～e[2]	65, 66, 67

多次元配列の初期化

多次元配列の初期化も、一次元配列のように初期値を指定します。その際、

——初期化リストは多次元配列の後ろの添字を先に変化させる順番で割り当てられる

という積み込みルールにしたがいます。

また初期値リストは次元ごとに{}で区切ることができます。こうすると次元境界を意識した割り当てができます。

次の例は同じ初期化を行ないます。a[0][0]は10に、a[1][0]は40になります。

例1

```
int a[2][3] = { 10, 20, 30, 40, 50, 60 };
```

例2

```
int a[2][3] = { {10, 20, 30}, {40, 50, 60} };
```

例3

```
int a[2][3] = {
        { 10, 20, 30 },
        { 40, 50, 60 },
    };
```

多次元配列のサイズ省略と初期値省略

多次元配列の先頭のサイズは省略できます。省略できるのは先頭の要素サイズだけです。

省略された先頭サイズは、初期値の数と2次元目以降の配列サイズから判断されます。

```
int a[][3] = {10, 20, 30, 40, 50, 60};     ——a[2][3]とみなされる
int a[2][] = {10, 20, 30, 40, 50, 60};     ——エラー
```

初期値の数が少ないときは、不足する要素は0になります。初期値の数が多すぎるときはエラーになります。初期値の全体数は正しくても、次元ごとの割り当て数が多すぎると、それもエラーになります。

```
int a[2][3] = { 10, 20, 30, 40 };    ——{10, 20, 30, 40, 0, 0}になる
int a[2][3] = { 10, 20, 30, 40, 50, 60, 70 };    ——エラー

int a[2][3] = { {10, 20}, {40} };    ——{{10, 20, 0}, {40, 0, 0}}になる
int a[2][3] = { {10, 20}, {30, 40, 50, 60} };    ——エラー
```

文字配列の初期化

char型の配列は、単に整数配列ですから、初期化方法はこれまで説明したのと同じです。設定内容が文字列条件を備えていれば、そのあとは文字列として利用できます。

また文字列リテラルで初期化することもできます。文字列リテラルを使った初期化で、配列サイズが省略されているときは、文字列長+1のサイズが設定されます（+1は¥0の分）。指定した配列サイズより、文字列リテラルの方が短いときは、残りは0が設定されます。

次の例はいずれも"ABCD"を設定しています。

```
char a[5] = { 'A', 'B', 'C', 'D', '¥0' };  ――文字定数で設定
char b[5] = { 65, 66, 67, 68, 0 };         ――数値で設定
char c[5] = "ABCD";                         ――文字列リテラルで設定
char d[] = "ABCD";                          ――d[5]とみなされる
char e[40] = "ABCD";                        ――残り部分は0になる
puts(a);                                    ――出力："ABCD"
```

二次元の文字配列を初期設定する場合は次のように記述します。

```
char days[4][10] = {
    "Sunday",
    "Monday",
    "Tuesday",
    "Wednesday"
};
puts(days[1]);    ――出力："Monday"
```

041 構造体・共用体・ポインタの初期化 初期化4

構造体、共用体、ポインタなども初期化することができます。その初期化の詳しい方法は、それぞれの章で説明します。ここでは初期化の簡単な例を示します。C99では自動記憶域期間をもつ構造体および共用体を、「式」で初期化できるようになりました。

構造体の初期化例

```
struct sdata {
    int x;
    int y;
};

struct sdata d = {100, 200};  ——d.xを100に、d.yを200に設定
```

共用体の初期化例

```
union udata {
    int  id;
    char cd;
};

union udata d = {1000};       ——d.idを1000に設定
```

ポインタの初期化例

```
char s[100];
char *p1 = s;         ——文字配列sの先頭アドレスをp1に設定
char *p2 = "abcde";   ——文字列"abcde"を確保しその先頭アドレスをp2に設定
int n, *ip = &n;      ——変数nのアドレスをipに設定
```

042 要素指示子を使う初期化

初期化5

C99では**要素指示子**で要素位置を指定して配列の初期化ができます。

[定数式]=初期値　——配列に使用
.メンバ名=初期値　——構造体と共用体に使用

次の例は「配列の3番目と8番目の値だけを設定したい、あとはデフォルト値でよい」という場合の記述例です。

```
int dt[10] = { [3]=123, [8]=456 };
```

次の記述は配列の最初の3つと最後の3つに初期値を設定しています。要素指示子でいったん位置を指定すると、後続する要素指示子のない初期値は、次位置以降に自然順序で設定されます。要素指示子を使って、すでに初期化している値を書き換えてもかまいません。

```
#define SIZ 10
int d1[SIZ] = { 10, 20, 30, [SIZ-3]=100, 200, 300 };
                // 内容 → { 10, 20, 30, 0, 0, 0, 0, 100, 200, 300 }
int d2[] = { 0, 1, 2, 3, [2]=20, 30, 40 };
                // 内容 → { 0, 1, 20, 30, 40 }
```

構造体の初期化は、基本的にはメンバの登場順に初期値を設定する必要があります。要素指示子を使うと任意のメンバを決め打ちで初期化できます。後続する初期値は、その要素に続く自然順序でメンバに割り当てられます。

```
struct sdata { int a, b, c, d, e; };
struct sdata d = { .b=11, .d=22 };
                // 内容 → { 0, 11, 0, 22, 0 }
```

共用体の初期化は、基本的には最初のメンバに初期値を設定する必要があります。要素指示子を使うと任意のメンバに対して初期値を設定できます。

```
union Utyp { int it; short int si; char ch; };
union Utyp d = { .si = 1000 };   // siで初期化する
```

第8章

演算子

C Quick Reference

043 演算子と真偽値

演算子1

数値や文字列に対する演算や、その他の目的の演算を行なうときには**演算子**を使います。

演算子のうち、関係演算子(<　<=　>=　>)、等価演算子(==　!=)、論理演算子(&&　||　!)などは、条件判定に利用されます。その条件判定の式では、条件が成立していることを「真」、成立していないことを「偽」といいます。また両者を合わせて**真偽値**といいます。

Cでは「a==b」のような条件判定も数値を返します。C言語での条件判定は次のように行なわれます。

- **条件用演算子(==や<など)の処理結果は、真なら1、偽なら0になる**
- **数値そのものを判定するときは、0なら偽、非0なら真とする**

次にいくつか特徴ある記述例を示します。

```
int n, a=10, b=20;
n = a==99;                 ——nは0
n = a==10;                 ——nは1
n = (a==10) + (b==20);     ——nは2

if (a==10) ～              ——真である
if (a) ～                  ——真である
if (0) ～                  ——偽である
if (1) ～                  ——真である
if (-5) ～                 ——真である
```

044 算術演算子

演算子2

演算子	説明	例
+	正符号（単項プラス演算子）	+a
-	負符号（単項マイナス演算子）	-a
*	乗算	a=b*c;
/	除算	a=b/c;
%	剰余	a=b%c;
+	加算	a=b+c;
-	減算	a=b-c;

算術演算子は数値を計算するものです。結果として数値を得ます。整数除算を行なうと剰余は切り捨てられます。また整数に対して%演算子を使うと、剰余を求めることができます。整数を0で除算するのは不正です。

045 関係演算子と等価演算子

演算子3

関係演算子	説明	例
<	小さい	if (a<b)
<=	小さいか等しい	if (a<=b)
>	大きい	if (a>b)
>=	大きいか等しい	if (a>=b)

等価演算子	説明	例
==	等しい	if (a==b)
!=	等しくない	if (a!=b)

関係演算子はふたつのオペランドの大小関係を判定します。**等価演算子**はふたつのオペランドの等価関係を判定します。関係が正しければ結果は1（真）に、正しくなければ0（偽）になります。

046 論理演算子

演算子 4

演算子	説明	例
!	否定	if (!a)
&&	論理積(かつ)	if (a==b && c==d)
||	論理和(または)	if (a==b || c==d)

論理演算子は真偽値を否定したり、複数の条件を組み合わせるときに使用します。優先順位は高い方から！ && ||です。

```
if (!a)                ——aが偽なら!aは真になる
if (a==5 && b==5)      ——aもbも5なら真
if (a==5 || b==5)      ——どちらか一方(または両方)が5なら真
```

&&と||は左から右に評価されます。途中で結果が確定すれば、そこで条件判定を中止します(ショートカット判定)。合致する確率の高い条件を最初に記述した方が処理効率がよくなります。すべての条件式が実行されるとは限らないことに注意が必要です。

```
if (a == b || (c=d) == 5)         ——a==bが真ならc=dの代入は実行しない
if (a == b && (c=d) == 5)         ——a==bが偽ならc=dの代入は実行しない
if (s!=NULL && strlen(s)>5)       ——sがNULLならはstrlen(s)は実行しない
```

047 増分演算子と減分演算子

演算子5

演算子	説明	例
++	1加算	++a; または a++;
--	1減算	--a; または a--;

増分演算子と**減分演算子**は1加算、1減算を行ないます。この演算子は次の形式をもっています。

増分演算子と減分演算子の形式

```
++a     ――前置増分演算子
a++     ――後置増分演算子
--a     ――前置減分演算子
a--     ――後置減分演算子
```

単項演算として単純に用いるときは、単なる数値1の加減処理と同じです。

```
a=a+1; a+=1; ++a; a++; ――すべて同じ結果になる
a=a-1; a-=1; --a; a--; ――すべて同じ結果になる
```

他の式と組み合わせて使うときは、

――前置型ははじめに1加減処理をする。後置型はあとで1加減処理をする

という動作をします。次に++演算子を使った記述例を示します。

記述	同結果になる別記述
a = ++b;	b = b + 1; a = b;
a = b++;	a = b; b = b + 1;
d[++n] = 10;	n = n + 1; d[n] = 10;
d[n++] = 10;	d[n] = 10; n = n + 1;
*p++ = *q++;	*p = *q; p = p + 1; q = q + 1;

048 ビット単位演算子

演算子6

演算子	説明	例
&	ビット単位の論理積（AND）	a=b & 0x7FFF;
|	ビット単位の論理和（OR）	a=b | 0x8000;
^	ビット単位の排他的論理和（XOR）	a=b ^ 0x000F;
~	ビット単位の補数（NOT）	a = ~a;
<<	ビット単位の左シフト	a=a << 2;
>>	ビット単位の右シフト	a=a >> 2;

ビット単位演算子はビット単位でデータ操作をするものです。操作には論理処理、否定、シフトがあります。この演算子の処理対象は整数でなくてはなりません。

演算子	ビット処理の方法	
&	AND処理	両方のビットが1のとき結果が1になる
|	OR処理	少なくともひとつのビットが1なら結果が1になる
^	XOR処理	両方のビットが異なるとき結果を1にする
~	NOT処理	ビットを反転する

```
int a, dt=0x55555555;
a = dt & 0x0000FFFF;    // AND処理 aは 00005555
a = dt | 0x0000FFFF;    // OR処理  aは 5555FFFF
a = dt ^ 0x0000FFFF;    // XOR処理 aは 5555AAAA
a = ~dt;                // NOT処理 aは AAAAAAAA
```

シフト演算は、指定した数だけビット位置を左右にシフトさせます。

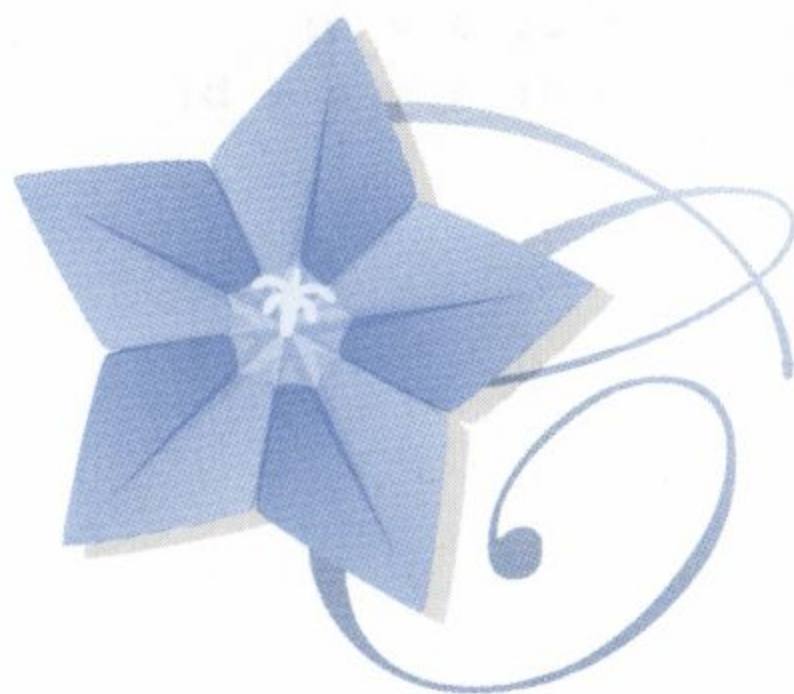
```
unsigned int ud, udt=0xABCDEF12;
int id, idt=0xABCDEF12, ID2=0x5678ABCD;
ud=udt; ud = ud << 4;  // (1) udは BCDEF120 左シフト(unsigned)
id=idt; id = id << 4;  // (2) idは BCDEF120 左シフト
ud=udt; ud = ud >> 4;  // (3) udは 0ABCDEF1 正値の右シフト(unsigned)
id=ID2; id = id >> 4;  // (4) idは 05678ABC 正値の右シフト
id=idt; id = id >> 4;  // (5) idは FABCDEF1 負値の算術右シフト
id=idt; id = id >> 4;  // (6) idは 0ABCDEF1 負値の論理右シフト
```

左シフトを行なうと左からはみだしたビットは捨てられ、右から0が補充されます。(1)(2)の記述が相当します。

右シフトを行なうと右からはみだしたビットは捨てられます。演算対象が正値の場合、左から0が補充されます。(3)(4)の記述が相当します。

右シフトで演算対象が負値の場合、C規格の定めはなく、次の2種類の動作があります。

算術右シフト：(5)の例に相当。左端の符号ビットと同じ値（例では1）を補充する
論理右シフト：(6)の例に相当。常に0を補充する

実際の処理系は算術シフトを採用しているのが一般的です。算術シフトは「算術計算に耐えるシフト」を意味しています。たとえば-1000を右1ビットシフトすると、2で整数除算した場合と同じ-500になります（奇数値の右シフトは要注意）。

049 代入演算子

演算子 7

演算子	種類
単純代入演算子	=
複合代入演算子	+= -= *= /= %= &= ^= |= <<= >>=

単純代入演算子と複合代入演算子

代入演算子は**単純代入演算子**と**複合代入演算子**のふたつの種類に分けられます。複合代入演算子の「A op= B」は「A = A op B」という式に相当します。

```
a = 10;          ――単純代入
a += 10;         ――複合代入。a = a + 10; と同じ
a *= b + c;      ――複合代入。a = a * (b + c); と同じ
```

多重代入式

代入演算子は**多重代入処理**することができます。次の記述はすべての変数に10を代入します。

```
a = b = c = 10;   ――このように書くことができる。a = (b = (c = 10)); と解釈
```

代入演算子の結合順序

多重代入処理がうまく働くのは、代入演算子の結合が右から左になっているからです。また優先順位の強さはすべて同じです。"+="の方が"="より強いということはありません。記述例を次に示します。

記述	解釈
a += b = c;	b = c; a = a + b;
a = b += c;	b = b + c; a = b;
a += b -= c;	b = b - c; a = a + b;

050 条件演算子

演算子8

演算子	説明	例
? :	条件処理	n = a>b ? 10 : 20;

条件演算子?:はif文と似た処理をコンパクトに表現する演算子です。「式1 ? 式2 : 式3」の形式で記述したとき、式1が真なら、式2を結果に、そうでなければ式3を結果にします。

```
int a = 50, n;
if (a>10) n = 20; else n = 30;       ——(1) if文で記述
n = a>10 ? 20 : 30;                  ——(2)同等の処理を条件式で記述
a>10 ? (n = 20) : (n = 30);          ——(3)別の記述方法
```

この例で(2)は、

——aが10より大なら20を、そうでないなら30を結果にする(ここまでが条件演算)。その結果をnに代入する

という処理をしています。

(3)は理解のために少し技巧的な記述を示したものです。結果は(2)と同等です。式2および式3の部分で代入処理をしているので、条件演算の結果である式値は捨てています。

Cの条件演算子?:は=より優先順位が高いので、(3)ではかっこを用いています。次の点に注意してください。なおC++では、この問題は解消されています。

```
a>10 ? n = 20 : n = 30;        ——(a)このように記述すると
(a>10 ? b = 20 : b) = 30;      ——こう解釈されてエラーになる
a>10 ? b = 20 : (b = 30);      ——こうすればとりあえず正しい
```

条件演算子を並べて書くこともできます。その場合は次のように解釈されます。

```
ch = n>30 ? 'A' : n>20 ? 'B' : n>10 ? 'C' : 'D';      ——この文は
ch = n>30 ? 'A' : (n>20 ? 'B' : (n>10 ? 'C' : 'D'));
                                                      ——このように解釈する
```

051 カンマ演算子

演算子9

演算子	説明	例
,	複数式の実行	a = 10, b = 20;

カンマ演算子（またはコンマ演算子）は本来ひとつしか式を書けないところに複数の式を書くことを可能にします。基本的な用法を示します。

```
a = 10, b = 20;    ——結果はa=10; b=20;と同じ
```

カンマ演算子を使うと、全体がひとつの式とみなされますから、「ひとつの式しか書けない位置に複数の式を記述する」ことを可能にします。

```
sum = 0;
for (n=2; n<=10; n+=2)                     } 2〜10までの偶数の和を計算
    sum += n;

for (sum=0,n=2; n<=10; sum+=n,n+=2)        } カンマ演算子を利用した記述
    ;

if (a < 0) { a=10; b=20; c=30; }           ——{}が必要
if (a < 0) a=10, b=20, c=30;               ——{}が不要になる
```

カンマ演算子を使った式は左から評価されます。そして最後の式の計算結果がこの式全体の式の値になります。

```
n = (a = 10, b = 20);                      ——aは10, bは20, nは20になる
```

関数の引数部分にカンマ演算子を使うときは、本来の関数区切りのカンマと区別するために、かっこで囲みます。

```
foo(10, (a=22,b=33), 40);                  ——この記述は
a=22; b=33; foo(10, 33, 40);               ——これと同じ結果になる
```

052 sizeof演算子

演算子形式	説明	例
sizeof 単項式	オブジェクトのサイズを返す	**n = sizeof dt;**
sizeof(型名)	データ型のサイズを返す	**n = sizeof(int);**

sizeof演算子は引数で示されたオブジェクトまたはデータ型のサイズを、バイト数で返します。単項式部分には一般の変数、配列、構造体、共用体の名前やオブジェクトを作る式などを書くことができます。たとえば"abcde"という文字列リテラルはオブジェクトを作る式です。

sizeof式の結果は実装依存のsize_t型(stddef.hなどで定義)です。これは符号なし整数(例:unsigned int)ですが、結果が小さいことが自明なときはint型の変数に代入すれば十分です。

次に例を示します。int型は4バイトであるとします。

```
char   cdt;
int    idt, ary[10], n;
double ddt;
struct typ { int a,b,c; };

n = sizeof cdt;             ――nは1
n = sizeof idt;             ――nは4
n = sizeof ddt;             ――nは8
n = sizeof ary;             ――nは40(10×4)
n = sizeof "abcde";         ――nは6(文字'¥0'をふくむ)
n = sizeof(char);           ――nは1
n = sizeof(short int);      ――nは2
n = sizeof(int);            ――nは4
n = sizeof(float);          ――nは4
n = sizeof(double);         ――nは8
n = sizeof(struct typ);     ――nは12になる
```

sizeof演算子はオペランドが式であっても、その式を「評価」することはありません。sizeofとともに「副作用を期待する式」を書くことはできません。C99では、オペランドの型が可変長配列型である場合は、オペランドを評価します。

```
int a, b = 100;
a = sizeof(++b);                   ――この++bは評価されない
printf("a=%d b=%d¥n", a, b);       ――出力:a=4 b=100
```

053 アドレス演算子と間接参照演算子 演算子11

演算子	説明
&	対象オペランドのアドレスを返す
*	ポインタの指している対象を間接参照する

アドレス演算子と**間接参照演算子**はポインタ処理に利用します。
詳しい説明は「第10章 ポインタ」(p. 97)にあります。
int型オブジェクトを処理する例を示します。

```
int obj, n;     // int型変数
int *p;         // int型オブジェクトを指すポインタ

p = &obj;       // objのアドレスをpに代入
*p = 123;       // pが指すオブジェクト(=obj)の内容を123にする
n = *p;         // 変数nは123になる(n=obj; を実行したのと同じ)
```

054 キャスト演算子 演算子12

キャスト演算子は式の結果を指定したデータ型に変換します。一般的な形式は次のとおりです。

キャスト演算子の書式

```
(型) 式
```

基本的なキャスト式の例を示します。

```
char ch;
int intDt;
double dblDt;
char *chrPt;
int *intPt;

intDt = (unsigned char)ch;      ――文字コードを符号なし型に変換
intDt = (int)dblDt;             ――double型の値をint型に変換
intPt = (int *)chrPt;           ――char*型ポインタをint*型に変換
```

055 メンバアクセス演算子

演算子13

演算子	説明
.	直接メンバ指定。構造体や共用体などのメンバを指定する
->	間接メンバ指定。同上（ポインタで指して場合に使用）

メンバアクセス演算子は構造体や共用体のメンバを指定するときに用います。

演算子'.'は構造体や共用体が通常のオブジェクト表現されたときのメンバ指定に用います。

演算子'->'は構造体や共用体を指すポインタを使う場合のメンバ指定に用います。

```
struct smp { int a, b; };
struct smp ob, *pt = &ob;
ob.a = 100;     ――オブジェクト名のときのメンバ指定
pt->b = 200;    ――ポインタのときのメンバ指定。(*pt).bと同じ
```

056 添字演算子

演算子 14

配列の要素を指すときに使う [] を**添字演算子**といいます。たとえば、

```
int dt[10];
dt[4] = 1234; ――dtの（0から数えて）4番目の要素に代入する
```

のように使います。添字演算子を使った添字式はC規格では次のように評価されます。

```
E1[E2]             ――この式は
*((E1)+(E2))       ――このように評価される
```

ここでE1およびE2のうち、一方はポインタ、他方は整数を使います。配列名はポインタに評価されます。E1とE2は、位置可変です。次の例はいずれも正しい記述です。

```
int n, a = 2, ary[] = {10, 20, 30, 40};
n = ary[a];     ――nは30
n = a[ary];     ――同上
n = ary[2];     ――同上
n = 2[ary];     ――同上
```

057 関数呼出演算子

演算子 15

関数呼出演算子は関数呼び出しを行ないます。関数へのポインタである式に () が後続する形式をもち、通常は「関数名 (引数並び) 」という利用をします。

void型以外の関数は、関数呼び出しの結果、その関数型の値を返します。

```
fnc1();           ――()が関数呼出演算子である
fnc2(10,20);      ――引数をともなう関数呼出演算子
n = fnc3(30);     ――関数呼び出しの結果の値を利用する
```

058 演算子の優先順位と結合規則 演算子16

優先順位

これまで説明した演算子には**優先順位**というものがあります。優先順位の高い演算子が先に計算されます。また()を使うと、かっこの内部を先に計算します。次に例を示します。このような優先順位(演算子の強さ)が、すべての演算子について決まっています。

```
n = 1 + 2 * 3;      ——⑴2*3を先に計算
n = (1 + 2) * 3;    ——⑵かっこの中を先に計算
if (a & b == c)     ——⑶間違いやすい例。「if (a & (b == c))」とみなす
```

結合規則

同じ優先順位の演算子が連続しているときは、**結合規則**にもとづいて優先評価されます。

```
n = 8 / 4 * 2;      ——⑴左から右に結合。8/4を先に実行
a = b = 10;         ——⑵右から左に結合。b=10を先に実行
```

⑴では/と*の優先順位は同じですが、左結合(左から右に結合)であるため「(8/4)*2」の計算が行なわれ、結果は4になります。これがもし右結合であれば、結果は「8/(4*2)」で1になるところです。

⑵では変数aとbの値はともに10になります。「b=10」を実行し、そのあと「a=b」を実行します。

演算子の優先順位と結合規則一覧を次ページに示します。この表で優先順位は枠単位で上の方が高くなります。結合規則の「左」は「左から右に結合していく」ことを意味します。「右」はその逆です。

演算子の優先順位と結合規則（上の方が優先順位が高い）

<table>
<tr><th>種類</th><th>演算子</th><th>結合規則</th></tr>
<tr><td>関数，配列，メンバ，後置++, --
複合リテラル（C99で追加）</td><td>() [] -> . ++ --
(型名){初期化子並び}</td><td>左</td></tr>
<tr><td>前置++, -- 単項</td><td>++ -- sizeof & * + - ~ !</td><td>右</td></tr>
<tr><td>キャスト</td><td>(型名)</td><td>右</td></tr>
<tr><td>乗除</td><td>* / %</td><td>左</td></tr>
<tr><td>加減</td><td>+ -</td><td>左</td></tr>
<tr><td>シフト</td><td><< >></td><td>左</td></tr>
<tr><td>比較</td><td>< <= > >=</td><td>左</td></tr>
<tr><td>等価</td><td>== !=</td><td>左</td></tr>
<tr><td>ビットAND</td><td>&</td><td>左</td></tr>
<tr><td>ビットXOR</td><td>^</td><td>左</td></tr>
<tr><td>ビットOR</td><td>|</td><td>左</td></tr>
<tr><td>論理AND</td><td>&&</td><td>左</td></tr>
<tr><td>論理OR</td><td>||</td><td>左</td></tr>
<tr><td>条件</td><td>?:</td><td>右</td></tr>
<tr><td>代入</td><td>= += -= *= /= %= &=
^= |= <<= >>=</td><td>右</td></tr>
<tr><td>カンマ</td><td>,</td><td>左</td></tr>
</table>

第9章 文

C Quick Reference

059 文の種類

文1

文には次のものがあります。文は逐次実行され、実行されることでなんらかの効果を残します。

文

ラベルつき文	
式文	
複合文	`{}`
選択文	`if switch`
繰り返し文	`for while do-while`
分岐文	`goto continue break return`

選択文、繰り返し文、分岐文は、**制御文**と呼ばれます。制御文は処理の流れを制御します。

このうち`if`, `while`, `for`, `do`, `switch`においては、制御式を評価し、結果の値が0以外なら真、0なら偽となり、指定された処理に移ります。制御式はかっこで囲みます。これらの制御文の制御範囲は「後続するひとつの文」です。もし被制御文が複数の文になるときは複合文`{}`を用います。

060 式文、空文、ラベルつき文

文2

式文

式文の書式

```
式opt ;
```

式文は「式」に`;`をつけたものです。「式」だけでは実行させることができませんが、'`;`'を付与すると、文という実行単位になります。

```
a = 100        ――これは式
a = 100;       ――これは式文
```

次のものも正しい式文です。実行させても無意味ですが、(警告は出るかもしれない)コンパイルエラーにはなりません。

```
10;          ┐
n;           ├ これも正しい式文
"abcd";      ┘
```

空文

式文で「式」を省略すると、**空文**(くうぶん)と呼ばれる特殊な文になります。空文は「その位置に文が必要だが、記述する内容はない」という場合に使います。たとえば制御文で空の本体を置くために使われたり、文がない位置にラベルを置くために使われたりします。

```
int n;
char s[] = "ABCDE";
for (n=0; s[n]; n++)
    ;          ──これが空文。nは文字列sの長さ5になる

if (a > b) {
    if (c == d) goto Lb1;
        ...
    Lb1: ;  ──'}'の直前で「文」がないので空文を置いている
}
```

ラベルつき文

ラベルつき文の書式

```
識別子 : 文
case 定数式 : 文
default : 文
```

Cには3つの**ラベルつき文**があります。「識別子」ラベルはgoto文のジャンプ先として用います。caseラベルとdefaultラベルはswitch文の中で使用します。ラベルつき文の用法は、goto文およびswitch文のところで説明します。これらのラベルは「文の前につく」ということに注意してください(上の例Lb1:参照)。

061 複合文

文3

複合文の書式

```
{ 宣言並び_opt 文並び_opt }
{ 宣言または文の並び_opt }      ――C99の規格
```

複合文（ブロック）は0個以上の宣言および、0個以上の文からなります。文を書くことのできる位置なら、どこにでも複合文を記述できます。複合文は通常は制御文と組み合わせて使用し、「ひとつの文しか書けない位置に複数の文を書く」という目的で使われます。内部に複数の文があっても{}で囲まれていれば、それ全体がひとつの文として処理されます。

また複合文は新しい有効範囲を作るために使用されることもあります。

複合文内で宣言した変数は、その複合文の内部のみで有効であり、また複合文の外で宣言されている名前より優先されます。

複合文内での宣言は、先頭部分（文より前）に記述します。C99では適切な任意の位置で宣言できます。→「014 変数宣言の位置」(p. 22)

```
if (a > b)
    c = 10;    ――非制御文をひとつ書く

if (a > b) {
    c = 10;  }
    d = 10;  } 複数の文を記述する
}

if (a > b) {
    int tmp;    ――複合文内のみで通用する変数を宣言
    tmp = a; a = b; b = tmp;
}

{
    int tmp;   } 単独の複合文を記述
    ...
}
```

062 if文

文4

if文の書式

```
if (式) 文             ——(1)
if (式) 文1 else 文2   ——(2)
```

if文は「式」の値によって分岐を行ないます。形式(1)の場合、「式」が真であれば、「文」を実行します。形式(2)の場合、「式」が真であれば「文1」を、偽であれば「文2」を実行します。

```
if (n < 0) {                    // if形式
    x = 10;
    y = 20;
}

if (a == b)                     // if-else形式
    puts("aとbは等しい");       // 真のとき実行
else
    puts("aとbは等しくない");   // 偽のとき実行
```

次のようにif-else if形式を用いることで、多方向分岐を行なうことができます。「else if」は必要なだけ記述できます。

```
if (n >= 90)                    // 点数が90点以上なら
    val = 'A';                  // 評価はA
else if (n >= 70)               // そうではなく70点以上なら
    val = 'B';                  // 評価はB
else if (n >= 50)               // そうではなく50点以上なら
    val = 'C';                  // 評価はC
else                            // それ例外は
    val = 'D';                  // 評価はD
```

if文においてelseは「elseの前でもっとも近い位置にあるifと結びつく」というルールで処理されます。対応があいまいな場合は、非制御文がひとつであっても、適切に{ }を用いると分かりやすくなります。

note 選択文と繰り返し文の新しいブロック有効範囲

C99では選択文と繰り返し文は、独自の**ブロック有効範囲**をもつものとして処理されるようになった。たとえば選択文の規則は、

——選択文はブロックとする。そのブロックの有効範囲は、それを囲むブロックの有効範囲の真部分集合とする。各副文もブロックとする。そのブロックの有効範囲は、選択文の有効範囲の真部分集合とする。

とある。これは複合リテラルの導入など、言語構造が複雑になることで、オブジェクトの有効範囲があいまいになることを防止するために意味がある。ブロックの存在を明確化すれば、オブジェクトのスコープ確定が唯一になり、異なった解釈の余地がなくなる。

```
if (a == '.') b = 10; else b = 20;                    // この文は
{ if (a == '.') { b =10; } else { b = 20;} }          // このように解釈する
```

063 while文

文5

while文の書式

```
while(式) 文
```

while文は「式」が真である間、「文」の処理を繰り返します。式の値がはじめから偽のときは、処理は一度も実行されません。

```
int sum = 0, n = 1;
while (n <= 5) {
    sum += n;           // sumの最終値は15
    ++n;
}
```

次のように書くと無限ループ処理になります。

```
while (1) {
    ...
    if (条件) break; ──while文を終了させる
}
```

064 do文

文6

do文の書式

```
do 文 while (式);
```

do文は、まず「文」を実行し、「式」が真であれば、繰り返し処理を続行します。式の値がはじめから偽のときも、最低一度は文が実行されます。

```
int sum = 0, n = 1;
do {
    sum += n;       // sumの最終値は15
    ++n;
} while (n <= 5);
```

065 for文

文7

for文の書式

for (式1_{opt}; 式2_{opt}; 式3_{opt}) 文
for (宣言 式2_{opt}; 式3_{opt}) 文 ——C99で追加。「宣言」という表現は';'をふくむ

for文は、最初の項が式1の場合、まずループ処理に入る前にその式1を実行します。次に式2が真の間、「文」の処理を繰り返します。毎回の文の処理のあとで「式3」を実行します。標準的な用法は、「指定した変数の値が値1から値2まで変化する間、ループ処理を行なう」というものです。

```
int sum = 0, n;
for (n=1; n<=5; n++) {
    sum += n;      // sumの最終値は15
}
```

この例で3つの式は制御用変数の管理をしていますが、そうではない自由な記述をすることもできます。もし意味のある記述なら、次の例でもかまいません。

```
for (a=b; c<d; e+=f) ...        ——正しい
```

3つの式はどれも省略できます。式2を省略すると、常に真とみなされ、無限ループ処理になります。

```
for (;;) {
    ...
    if (条件) break;            ——for文を終了させる
}
```

C99規格では式1の部分で変数宣言をすることができます。ここで宣言した変数は、このforブロックの中だけで有効です。

```
for (int n=1; n<=10; n++) {    ——nを宣言し初期値を与えている
    ...
}
```

066 switch文

文8

switch文の書式

```
switch (式) 文
```

caseラベルの書式

```
case 定数式: 文
```

defaultラベルの書式

```
default: 文
```

switch文は「式」で示す整数値によって多方向分岐します。「式」には整数を結果とするものを書きます。「文」は通常は複合文で指定し、内部に任意個のcaseラベル、および1個以下のdefaultラベルを記述することができます。caseラベルで用いる「定数式」の値は、重複してはいけません。

上記の構文を組み合わせた、switch文の標準的な形式を示します。

```
switch (式) {
case 定数式1:
      文並び
      break;          } 必要なcaseを並べる
case 定数式2:
      文並び
      break;
...
default:              } 省略可
      文並び
}
```

switch文の「式」の値が、どれかのcaseラベルの「定数式」の値と等しければ、そのcaseラベル位置の「文」に制御が移ります。どのcase定数式とも一致しないときは、defaultラベルがあればその位置に制御を移し、ないときはswitch文を終了します。

どれかのcaseラベルに制御が移ると、その位置以降の文を次つぎと実行します。そしてbreak文に出合うと、または末尾の}に出合うとswitch文を終了します。したがって通常はcaseラベルとbreak文をペアで用いるのが一般的です。

次に記述例を示します。

```
int n;
for (n=1; n<=4; n++) {
    printf("%d:", n);
    switch (n) {
    case 1:
        puts("値は1");
        break;    // (注)
    case 3:
        puts("値は3");
        break;
    default:
        puts("その他");
    }
}
```

実行結果

```
1:値は1
2:その他
3:値は3
4:その他
```

参考：(注)位置のbreakがない場合の実行結果

```
1:値は1
値は3
2:その他
3:値は3
4:その他
```

067 break文、continue文、goto文 文9

break文とcontinue文

break文の書式

```
break ;
```

continue文の書式

```
continue ;
```

break文はfor文、while文、do文、そしてswitch文の中で用いられ、制御を終了するのに用いられます。

continue文はfor文、while文、do文の中で用いられ、ループ処理のうち、その回の処理をパスします。実行は次のループ処理に(for文の場合は式3に)移ります。breakとcontinueの動作を次に示します。

```
int a;
for (a=1; a<=10; a++) {
    if (a == 5) break;          // 終了する
    printf("¥na=%d: start ", a);
    if (a == 2) continue;       // 次のループに移る
    printf("end");
}
```

実行結果

```
a=1: start end
a=2: start   ──continueの効果で"end"出力がパスされている
a=3: start end
a=4: start end
            ←──breakの効果でループ終了になっている
```

対象となる制御文がネストしている場合、break文とcontinue文は、それがふくまれるもっとも内側の制御文に対して作用します。またswitch文中のbreak文は、caseからはじまる処理を終了するのに使われます。

goto文

goto文の書式

```
goto ラベル名 ;
```

ラベル名の記述

```
識別子 : 文
```

goto文は対応する**ラベル名**のついた文（ラベルつき文）に処理を移します。同じ関数内の特定の文に強制的にジャンプするときに用います。ラベルは対応するgoto文より、前にあっても、後ろにあってもかまいません。次の例はgoto文の機能を確認するものです。

```
void foo(void)
{
    puts("aaa");
    goto asoko;      ─┐
    puts("bbb");      │ ジャンプする
asoko: ←─────────────┘
    puts("ccc");
}
```

実行結果

```
aaa ccc
```

ラベルは任意の文の頭につくものなので、もし関数の終端部にジャンプさせたいときは、あえて空文を置く必要があります。

```
void func(void)
{
    ...
    if (条件) goto owari;
    ...
owari: ;  ――空文';'を置き、その前にラベルをつける
}
```

068 return文

文10

return文の書式

```
return 式opt ;
```

return文は現在の関数を終了して、呼び出し側に制御を戻します。「式」があるときは、その値を関数戻り値にします。

「式」のないreturn文は、void型関数の中で使用します。「式」のあるreturn文は、非void型関数の中で使用します。

void型関数の中では、return文は必須ではありません。return文がないときは、関数末尾に到達すると関数処理を終了します。

```
int maxdt(int a, int b)      ――大きい値を返す関数
{
    if (a > b) return a;
    else       return b;
}

void disp_sqrt(double d)     ――正の数なら平方根を表示する関数
{
    if (d < 0.0) return;
    printf("平方根=%f¥n", sqrt(d));
}
```

note return文の値有無の厳密化

従来のC規格で、次の関数記述は（警告はされるが）正しくコンパイルできた。C99では、式をもたない**return**文は**void**型関数の中にしか記述できなくなった。

```
int foo(void)  // 戻り値はint型だが
{
    ...
    return; // 式のないreturn文を使っている。[C99]では不正
}
```

第10章

ポインタ

C Quick Reference

069 ポインタの基本用法1

ポインタ1

ポインタ型の宣言

オブジェクトはメモリ上に配置されたアドレスをもっています。また関数は実行開始のアドレスをもっています。**ポインタ**はこれらのアドレスを保持できる機能です。ポインタ型の宣言は次のように行ないます。

```
char *cp;       ──cpはchar型オブジェクトへのポインタ
int *ip;        ──ipはint型オブジェクトへのポインタ
int (*fp)();    ──fpはint型を返す関数へのポインタ
```

ここでfpは関数ポインタです。これについては、あとでまとめて説明します。

ポインタ宣言が、

```
T *P;
```

で表現できるとき、Pは「T型へのポインタ」と表現されます。なお、*の前後の空白は自由なので、次のいずれのスタイルで書いてもかまいません。Cでは(1)がよく使われます。C++では(2)もよく使われます。

ポインタ宣言のスタイル

```
(1)int *pt;
(2)int* pt;
(3)int * pt;
(4)int*pt;
```

ポインタは宣言時に型を指定しますが、この型により、現在保持しているアドレスから何バイトを処理対象にするかが決定されます。たとえば次のようになります。int型は32ビット幅とします。

```
char *cp;       ──現在のアドレスから1バイトが処理対象
int *ip;        ──現在のアドレスから4バイトが処理対象
double *dp;     ──現在のアドレスから8バイトが処理対象
```

アドレス演算子と間接参照演算子を使う

ポインタに関して、アドレス取得および間接参照をする次の演算子があります。

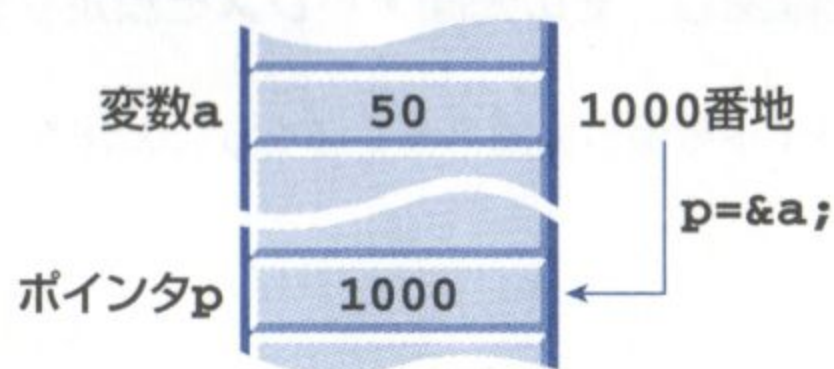
& ──対象オペランドのアドレスを返す

***** ──対象オペランドを間接参照する

これらの演算子の利用例を次に示します。

```
char a, b;      // オブジェクト
char *p;        // ポインタ

p = &a;         // aのアドレスを設定
*p = 50;        // 対象アドレスにある値を50にする(aが50になる)
b = *p;         // bは50になる

printf("%d %d %d ¥n", a, b, *p);    // 出力:50 50 50
```

この例で変数aが1000番地に確保されているとすると、ポインタpの内容は1000になり、*pは「1000番地にあるオブジェクトを示す左辺値」になります。このとき「*p」という表現は、実質「a」と同等です。どちらも同一のオブジェクトを表現しています。

変数とポインタの違いの概念図

070 ポインタの基本用法2

ポインタ2

ポインタの設定

ポインタはアドレスを保持するものです。そのアドレスは代入処理で設定できます。代表的なポインタ設定の例を次に示します。

例1：変数のアドレスを＆演算子で代入する

```
int d, *p;
p = &d;
```

例2：配列の先頭アドレスを代入する。配列名sを単独で指定すると、配列全体の先頭アドレスに評価される

```
char s[80], *p;
p = s;
```

例3：配列の1要素のアドレスを＆演算子で代入する

```
char s[80], *p;
p = &s[3];      // 添字3番目の要素のアドレス
p = &s[0];      // これはp=s;と同じ結果になる
```

例4：char型ポインタに文字列リテラルの先頭アドレスを設定する

```
char *p;
p = "abcd";
```

例5：プログラム実行開始後、動的にメモリを確保し、その先頭アドレスを設定する

```
char *p;
p = (char *)malloc(100);    ――100バイト確保し先頭アドレスをpに入れる
```

ポインタと整数の相互代入

ポインタと整数はそのままでは相互代入可能ではありませんが、定数値0だけはポインタにそのまま代入してよく、ポインタとの比較もできます。

また定数値0の代わりに記号定数NULLが用意されており、これを使うこともできます。NULLはstdio.hの中で0または((void *)0)と定義されています。

```
char *p;
p = 0;        ――0を設定する
p = NULL;     ――NULLを設定する
```

NULL(または0)が設定されたポインタを「空ポインタ」といいます。空ポインタは、

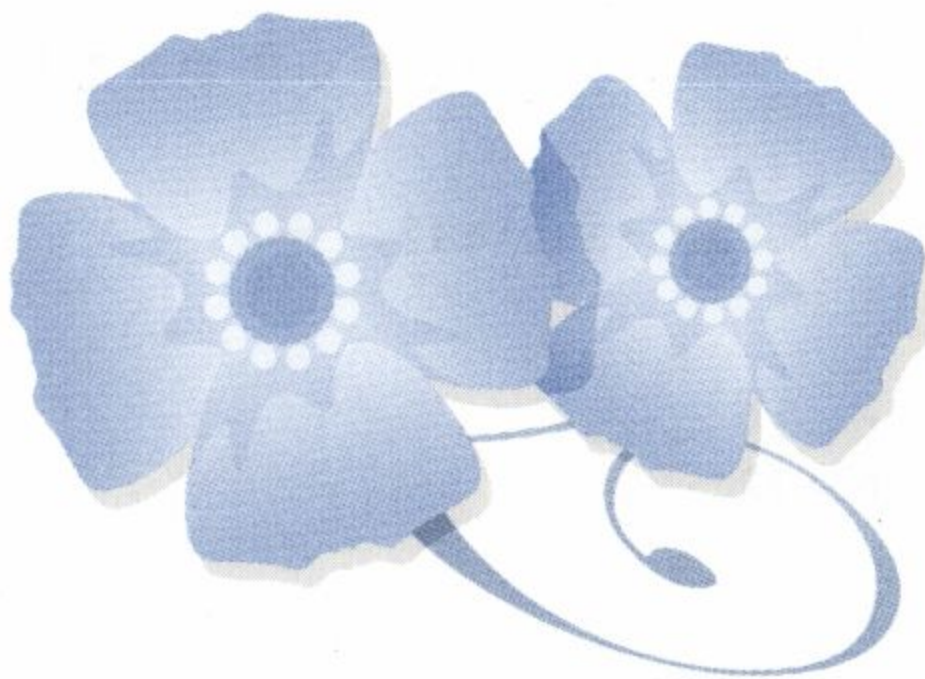
――意味のあるアドレスがまだ設定されていない

——ポインタが特別な意味にある(多くはエラー発生を伝える)

などの状態であることを示します。たとえばポインタチェーンの終端マークに用いたり、ポインタを返す関数でエラー発生を伝えるために使います。次に記述例を示します。

```
FILE *fp;
if ((fp=fopen("myfile.txt","r")) == NULL)    // NULLならオープン失敗
```

ポインタと整数は、適切にキャストすれば相互代入できます。ポインタの大きさと整数の大きさが同じなら、情報を失いません。キャストした整数をポインタに代入したとき、そのアドレスが正しいものであることを保証するのはプログラマの責任です。

```
int a, b, *ip;
ip = &a;                        // aのアドレスを設定
b = (int)ip;                    // 整数変換して代入
ip =(int *)b;                   // ポインタ変換して代入
printf("%08X %p¥n", b, ip);     // 出力例：0012FF3C 0012FF3C
ip =(int *)0x12FF3C;            // 整数定数を直接記述した例
```

071 ポインタの基本用法3

ポインタ3

ポインタの初期化

ポインタ宣言のときに初期化を行なうことができます。このとき用いる初期値は通常、コンパイラが確保した意味のあるアドレスを使います。いろいろな初期化の例を示します。最後の例は、プログラム実行開始後、動的にメモリを確保し、その先頭アドレスを設定します。この記述はグローバル変数の初期化には使えません。

```
int d;
int *p1 = &d;                       ——単純変数のアドレスを設定する

char s[80];
char *p2 = s;                       ——配列の先頭アドレスを設定
char *p3 = &s[0];                   ——同上。これでも配列の先頭アドレス設定になる
char *p4 = &s[3];                   ——s[3]のアドレスを設定
char *p5 = "abcde";                 ——文字列リテラルの先頭アドレスを設定
char *p6 = (char *)malloc(100);     ——メモリ領域を確保し先頭アドレスを設定
```

ポインタの演算

ポインタは代入処理(=)、単項&演算、単項*演算以外に次のような演算を行なうことができます。

- ポインタと整数の加算(+)と減算(-)
- ポインタ同士の減算(-)
- ポインタの増分、減分処理(++ --)
- ポインタ同士またはポインタと0の等価判定(== !=)
- ポインタ同士の比較判定(< <= > >=)

■①ポインタと整数の加算・減算およびポインタの増分・減分処理

ポインタと整数との間で、加算や減算の処理ができます。1加算すると、ひとつ次の要素を指します。

また++, --演算子を使うと、ポインタ自身を更新することができます。

```
int n, ary[5] = {10, 11, 12, 13, 14}, *p

p = ary;
n = *p;           // nは10
n = *(p + 2);     // nは12
p = p + 2;
n = *p;           // nは12
```

```
++p;
n = *p;              // nは13
--p;
n = *p;              // nは12
p = p - 2;
n = *p;              // nは10
```

ポインタを1加算すると、アドレスが1番地進むのではなく、「1要素分進む」ので注意してください。次の例でchar型ポインタは1番地進み、int型ポインタは4番地進んでいます(32ビットint型の場合)。

```
char ss[] = "ABCDE", *chrp;
int dd[] = {10, 11, 12}, *intp;
chrp = ss; printf("%p¥n", chrp);    // 出力例：0012FEE4
++chrp;    printf("%p¥n", chrp);    // 出力例：0012FEE5
intp = dd; printf("%p¥n", intp);    // 出力例：0012FF08
++intp;    printf("%p¥n", intp);    // 出力例：0012FF0C
```

■②ポインタ同士の減算処理

ポインタとポインタ間で減算ができます。この減算処理は両方のポインタが、同じ配列の要素を指しているときに実用的です。

```
p1 = ary;            // ary[0]のアドレス
p2 = ary + 2;        // ary[2]のアドレス
n = p2 - p1;         // 減算実行。nは2(要素単位の差)
```

■③ポインタの等価、比較判定

ポインタ同士での等価判定ができます。またポインタと数値0(NULL)との等価判定ができます。

ポインタ同士では大小比較判定もできます。比較判定は、両方のポインタが同じ配列の要素を指しているときに実用的です。

```
p1 = ary;
p2 = ary + 2;
if (p1 == 0)    n = 1; else n = 0;     // nは0
if (p1 == NULL) n = 1; else n = 0;     // nは0
if (p1 == p2)   n = 1; else n = 0;     // nは0
if (p1 < p2)    n = 1; else n = 0;     // nは1
```

072 ポインタと配列

ポインタ4

ポインタ表現と配列表現

ポインタと配列は密接な関係があります。ポインタを使って、配列風の表現ができます。逆に配列名を使ってポインタ的表現ができます。

配列名は、sizeof演算子のオペランドや、単項&演算子のオペランドなど、特殊な場合を除いて、確保配列の先頭アドレスを示すポインタとして評価されます。

```
char ch, s[80] = "ABCDE", *p;
p  = s;          ——(1)配列名sは「char型へのポインタ」
ch = s[2];       ——(2)chは'C'
ch = p[2];       ——(3)同上

ch = *p;         ——(4)chは'A'
ch = *s;         ——(5)同上
ch = *(p+2);     ——(6)chは'C'
ch = *(s+2);     ——(7)同上
```

この例でsは「char型へのポインタ」であり、「char型データ80個の先頭アドレス」を表現します。(1)は配列名を使って、pにアドレスを設定しています。(2)は標準的な配列表現ですが、式としては、

ポインタ[整数]

という指定になります。したがってポインタとして評価されるsを、pに置き換えれば、(3)も正当な記述になります。

(4)(6)はポインタを使った標準的な記述です。(5)(7)は配列名を使ったポインタ風の記述です。

名前と添字は位置互換

「056 添字演算子」(p. 80)でも説明しましたがポインタや配列名を使った添字式E1[E2]は、

```
*((E1)+(E2))     ——一方がポインタ、他方が整数
```

と処理されるので、名前(ポインタ)と添字は位置互換です。次の記述は違和感がありますが、正当なものです。

```
char ch, s[] = "ABCDE", *p = s;
ch = s[2];          ――chは'C'
ch = 2[s];          ――同上
ch = p[2];          ――同上
ch = 2[p];          ――同上
ch = "abcde"[2];    ――chは'c'
ch = 2["abcde"];    ――同上
```

void型ポインタの操作

void型ポインタは特殊なポインタで、アドレスの格納だけができます。++や--演算などはできません。void型ポインタは要素のサイズをもっていないので、これらの計算は不可能であり、また意味もありません。

優先順位の問題

ポインタ処理では++演算子や+演算子などを多用します。このとき計算の優先順位を誤りやすいので注意が必要です。たとえば文字配列コピーの定石記法である、

```
while (*a++ = *b++)
    ;
```

といった使い方のときに混乱が生じがちですから気をつけてください。そのルールの基本は、

――単項&演算子と単項*演算子は、算術演算子より結合が強い
――単項演算子は右から左に計算される

ということです。具体的な記述パターンを次に示します。解釈のしかたがあいまいなときはかっこや空白文字を意図的に用いた方が安全です。

記述	その解釈	コメント
`*p+1`	`(*p)+1`	値を+1する
`*(p+1)`	―	番地を+1する
`*p+=1`	`(*p)+=1`	`*p`を+1する。`*p += 1`と書いた方がよい
`*p++`	`*(p++)`	番地を1だけ後加算する。`*(p++)`と書いた方が明解だが`*p++`が定石記法として多用される
`(*p)++`	―	値を1だけ後加算する
`*++p`	`*(++p)`	番地を1だけ先加算する。`*(++p)`と書いた方がよい
`++*p`	`++(*p)`	値を1だけ先加算する。`++(*p)`と書いた方がよい

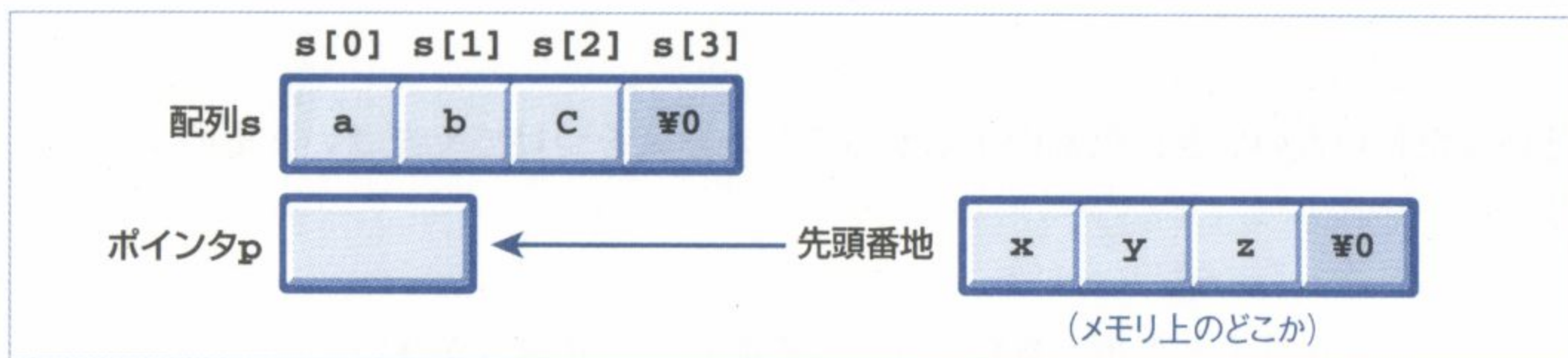
073 ポインタと文字列

ポインタ5

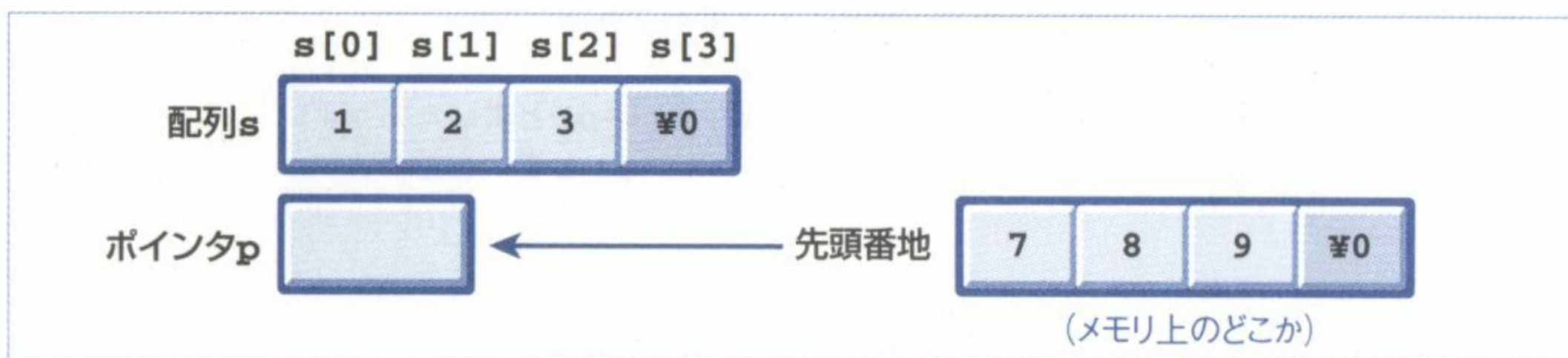
Cには「文字列」を表現する専用の機能はなく、その表現のためにはchar型配列を使います。ポインタと配列との関係を理解しておけば、ポインタで文字列を操作することは理解できます。しかしchar型配列には文字列実体を格納できる、ポインタはアドレスを格納するだけ、という違いを知っておく必要があります。ここではその違いを整理します。

■①ポインタの文字列初期化

```
char s[4] = "abc";     ——(1)
char *p = "xyz";       ——(2)
```

これは文字列リテラルによる初期化の違いです。(1)で配列sには文字列"abc"そのものが入ります。(2)の処理は、まず"xyz"という文字列実体をメモリ上のどこかに用意し、その先頭番地をpに代入します。

■②文字列コピーとポインタコピー

```
strcpy(s, "123");      ——(1)
p = "789";             ——(2)
```

これは初期化の場合と同じです。(1)で配列sには文字列"123"そのものが設定されます。(2)の処理は、まず"789"という文字列実体をメモリ上のどこかに用意し、その先頭番地をpに代入します。

■③ポインタによる文字列コピーとアドレスコピー

```
char s1[10], s2[] = "ABC", *p1 = s1, *p2 = s2;
strcpy(p1, p2);      ――(1)
p1 = p2;             ――(2)
```

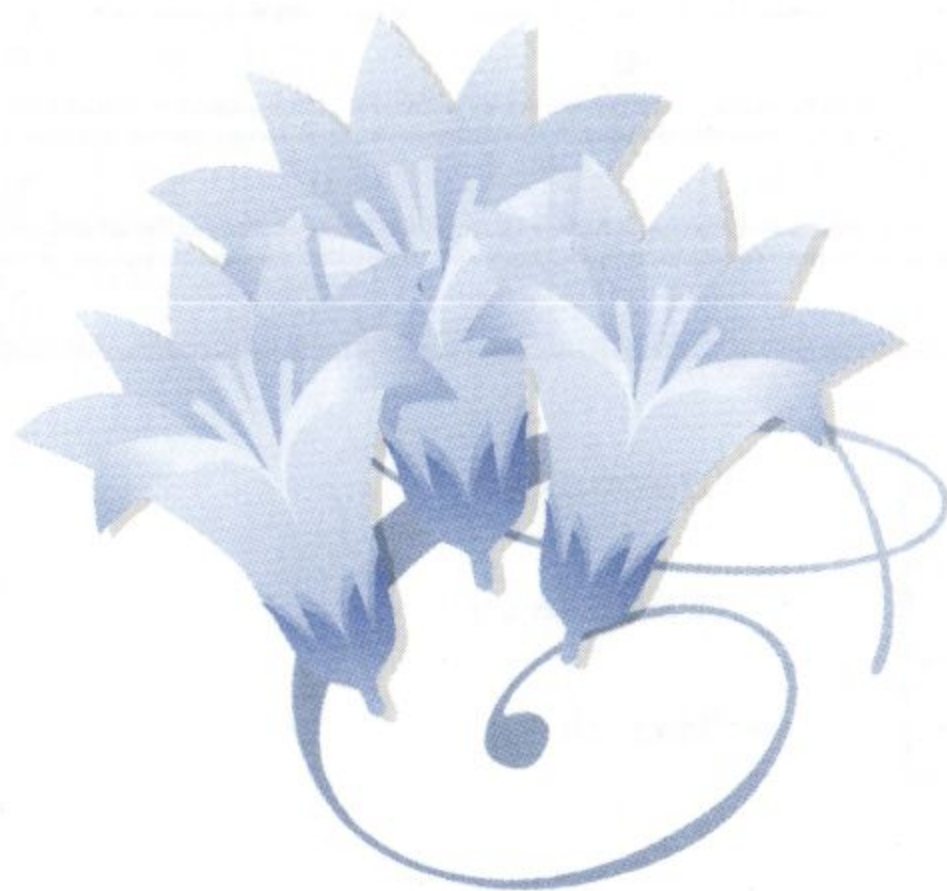
(1)は、p1が指すオブジェクト(s1)に、p2が指す文字列実体(s2)をコピーします。(2)のポインタコピーでは、p1に、p2のもつアドレスがコピーされ、以降はp1もp2も、s2を指すことになります。

文字列をポインタで管理すると、文字列実体を移動させることなく、つなぎ替え管理をすることができます。これを利用すると文字列処理を効率よく行なうことが可能です。たとえば文字列をソートする場合、メモリ上の文字列実体はそのままにして、代わりにポインタをつなぎ替えることで高速な処理ができます。

ポインタをつなぎ替える例を示します。

```
char s1[] = "AAA", s2[] = "BBB", *p1, *p2, *tmp;

p1 = s1; p2 = s2;
printf("p1=%s p2=%s¥n", p1, p2);    // 出力：p1=AAA p2=BBB
tmp = p1; p1 = p2; p2 = tmp;
printf("p1=%s p2=%s¥n", p1, p2);    // 出力：p1=BBB p2=AAA
```

074 ポインタの配列

ポインタ6

ポインタは配列にすることができます。これは多くの文字列を表現するときに便利です。ポインタ配列は、

```
char *pp[10];
```

のように宣言します。char型配列では多次元文字列の設定を次のように行なっていました。

```
char namev[3][10] = { "January", "February", "March" };
puts(namev[2]);        // 出力：March
```

似たようなこと（同一ではない）をポインタ配列では次のように行ないます。

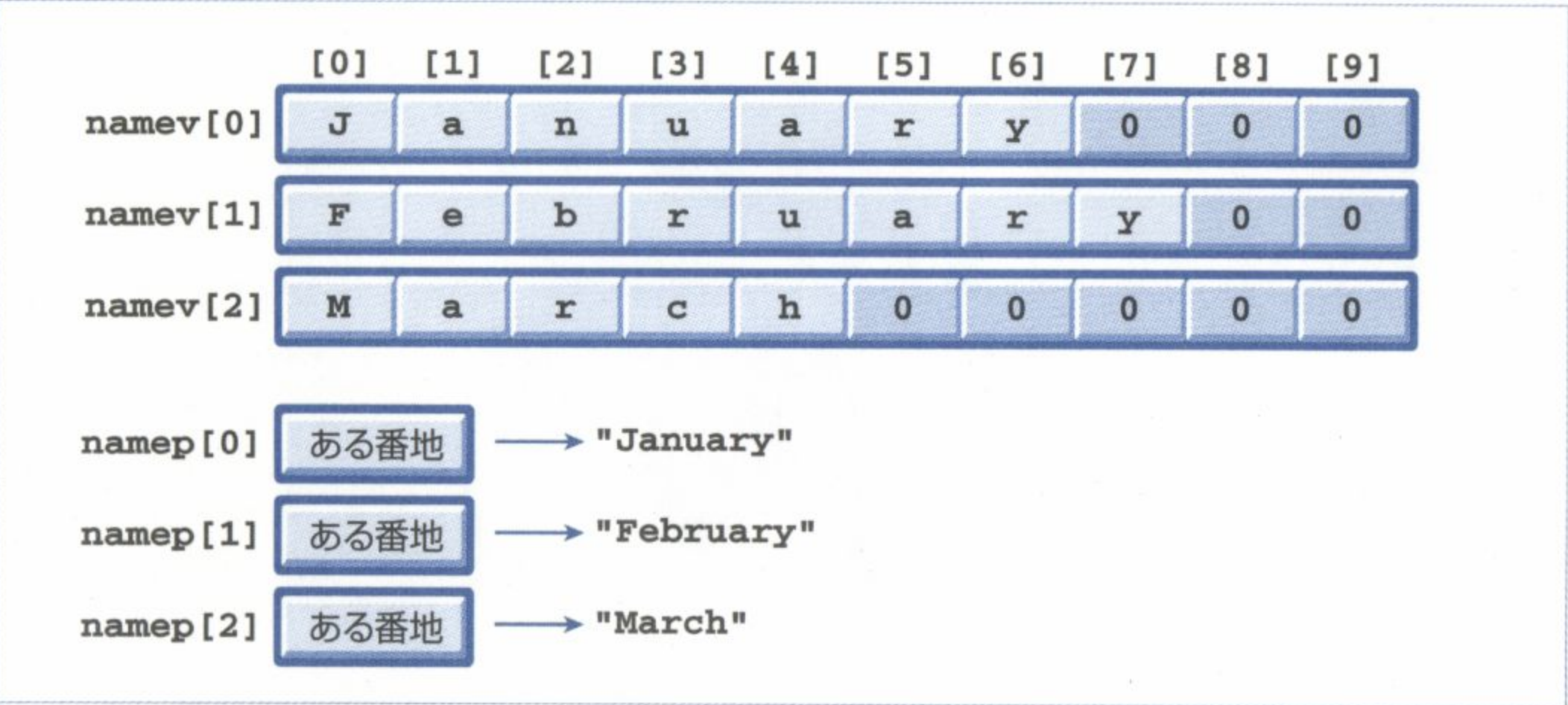
```
char *namep[3] = { "January", "February", "March" };
puts(namep[2]);        // 出力：March
```

両者はメモリ内のデータ確保の方法が異なります。配列ではそれぞれの配列要素（長さ10）に、データの実体が入ります。ポインタでは3つの文字列をメモリ上のどこかに確保し、その先頭番地をそれぞれのポインタに入れます。

両者の関係を次に図示します。

ポインタと配列による文字列確保の違い

075 ポインタのポインタ

ポインタ7

ポインタのポインタを宣言することができます。たとえば次のものです。

```
char **pp;
```

これは「char型の値を指すポインタを指すポインタ」です。次にポインタの宣言の違いをまとめます。

変数宣言とポインタ宣言の意味

宣言	説明
char a;	**a**は**char**型の変数
char *b;	「***b**」という記述が**char**型 **b**は**char**型(***b**)へのポインタ
char **c;	「****c**」という記述が**char**型 「***c**」という記述は**char**型(****c**)へのポインタ **c**は**char**型(****c**)へのポインタ(***c**)へのポインタ

ポインタのポインタは、さまざまに技巧的な利用ができますが、基本的には、

——ポインタは通常の配列を管理する
——ポインタのポインタは、ポインタの配列を管理する

という利用に便利です。次に例を示します。

```
char *p, **pp;
char ss[] = "ABC";
char *month[3] = { "January", "February", "March" };

for (p=ss; *p; p++)
    putchar(*p);
putchar('¥n');
for (pp=month; pp<month+3; pp++)
    puts(*pp);
```

実行結果

```
ABC
January
February
March
```

076 関数を指すポインタ1

ポインタ8

関数を指すポインタを宣言することができます。関数は実行コードの入口というアドレスをもっています。このアドレスをポインタに格納します。このあと、ポインタを使って関数を呼び出すことができます。

関数へのポインタは次のように宣言します。

```
int fnc(int n);          ——(1) int型を返す関数fnc
int *fnc(int n);         ——(2) int型へのポインタを返す関数fnc
int (*fnc)(int n);       ——(3) int型を返す関数へのポインタであるfnc
```

ここで(1)(2)は比較のために示したもので、一般的な表現です。(3)が関数へのポインタを宣言しています。(3)で*fncを囲むかっこは必要です。これがないと、識別子fncと続く'('との結合が強いので、(2)のように解釈されます。次に結合の強さを図示します。

```
int *  fnc(int n)  ;         ——(2)の結合関係

int   (*fnc) (int n) ;       ——(3)の結合関係
```

(3)は「*fnc」という表現が、「int型を返す関数」になり、それなら*を外したfncは「int型を返す関数へのポインタ」になるということです。

関数へのポインタを使ったプログラム例を示します。この例でcalcは、
——ふたつのint型引数をもち、int型を返す関数へのポインタ
という意味になります。なお、引数の名前は省略できるので関数原型は、

```
int (*calc)(int, int);  ——引数は型だけ指定
```

と書いてもかまいません。

関数へのポインタの使用例

```
#include <stdio.h>

int (*calc)(int a, int b);    // これが関数へのポインタの宣言
int add(int a, int b);        // これは普通の関数プロトタイプ
```

```
int sub(int a, int b);          // これも普通の関数プロトタイプ

int add(int a, int b)
{
    return a+b;
}

int sub(int a, int b)
{
    return a-b;
}

int main(void)
{
    int n;
    calc = add;                 // add()関数のアドレスを設定
    n = calc(30, 20);           // 関数へのポインタを使って関数呼び出し
    printf("%d¥n", n);          // 出力：50
    calc = sub;                 // sub()関数のアドレスを設定
    n = calc(30, 20);           // 関数へのポインタを使って関数呼び出し
    printf("%d¥n", n);          // 出力：10
    return 0;
}
```

note 関数指示子と&演算子と*演算子

次の(1)は(2)のように、(3)は(4)のように記述することも可能である。

```
calc = add;             ――(1)
calc = &add;            ――(2)
n = calc(30, 20);       ――(3)
n = (*calc)(30, 20);    ――(4)
```

関数指示子(function designator)は、関数型をもつ式であり、**sizeof**演算子または単項**&**演算子のオペランドである場合を除外して、「○型を返す関数」をもつ関数指示子は、「○型を返す関数へのポインタ」をもつ式に変換される。

たとえば関数指示子**add**は、(1)の形式で用いられると、関数のアドレス(厳密には「〜型を返す関数へのポインタ」)として暗黙に評価される。また(2)のように**&**演算子とともに用いられると、そのまま関数のアドレスを返す。結果的に(1)(2)は同じことになる。

Cでは関数呼び出しは「関数へのアドレス(引数)」で行なう。関数指示子をそのまま使って**add(〜)**とした場合、**add**は暗黙のうちに関数のアドレスに評価される。

(3)で**calc**は関数のアドレスであるから正当である。(4)の「***calc**」は関数アドレスからの間接参照で「関数指示子」になる。しかし関数指示子は暗黙のうちに関数のアドレスに変換されるから、(3)と(4)は同じになる。

Cでは関数指示子の暗黙評価があるから、文法的に次の記述はすべて正しいことになる。

```
n=add(1,2);  n=(&add)(1,2);  n=(*add)(1,2);  n=(**add)(1,2);
n=(***add)(1,2);
```

077 関数を指すポインタ2

ポインタ9

関数へのポインタの配列

関数へのポインタは配列にすることもできます。その配列要素はいくつもの関数の入口アドレスをもちます。たとえば、

```
void (*keisan[2])(int a, int b);
```

で、keisanは、

――ふたつのint型引数をもち戻り値がvoidである関数、への入口アドレスをふたつもつことができるポインタ配列

になります。

次のプログラムはそのポインタkeisanに、add, subというふたつの関数を割り当てたものです。なおここでは関数へのポインタを、main()関数の内部でローカル宣言しています。関数の内部で宣言するか外部で宣言するかは、変数宣言の場合と同じで、通用範囲が異なるだけです。

関数へのポインタの配列を見る

```
#include <stdio.h>
void add(int a, int b);
void sub(int a, int b);

int main(void)
{
    int i;
    void (*keisan[2])(int a, int b);   // 関数へのポインタの配列を宣言

    keisan[0] = add;                   // ┐ふたつの関数のアドレスを設定する
    keisan[1] = sub;                   // ┘

    for (i=0; i<2; i++)
        keisan[i](100, 20);            // ふたつの関数を順に呼び出す
    return 0;
}

void add(int a, int b)
{
    printf("add=%d¥n", a + b);
}

void sub(int a, int b)
{
    printf("sub=%d¥n", a - b);
}
```

実行結果

```
add=120
sub=80
```

関数へのポインタを引数にする

関数へのポインタは、関数への引数渡しに用いることもできます。そのよい例は標準ライブラリ関数のひとつである、qsort()関数です。qsort()関数の形式は、次のようになってます。

```
qsort(void *base, size_t n, size_t size,
                  int (*cmp)(const void *a, const void *b))
```

ここで(*cmp)関数は、ふたつのデータの大小比較を行なうものです。この関数を引数として供給することで、qsort()関数が汎用ソート関数になります。たとえば数値データソートも文字列データソートも可能になります。

簡単なモデルをあげておきます。次のプログラムは先にあげた加減計算のプログラムを「関数のポインタを引数にする」というスタイルに変更したものです。ここでkeisanは、

——ふたつのint型引数をとり、戻り値はない関数へのポインタを最初の引数にし、int型の第2引数と、int型の第3引数をもつ、戻り値のない関数

という意味になります。なお、関数keisan()は、基本どおりの記述では、

```
void keisan(void (*calc)(int a, int b), int dt1, int dt2); // 関数原型
void keisan(void (*calc)(int a, int b), int dt1, int dt2)  // 関数本体
{
     ...
}
```

となりますが、C規格では、

```
void keisan(void calc(int a, int b), int dt1, int dt2);
void keisan(void calc(int a, int b), int dt1, int dt2)
{
    ...
}
```

と簡潔に書くこともできます。次の例では簡潔な記述方法を用いています。

関数へのポインタを引数にしたプログラム

```
#include <stdio.h>
void keisan(void calc(int a, int b), int d1, int d2);
                                        // 関数プロトタイプ
void add(int a, int b);
void sub(int a, int b);

int main(void)
{
    keisan(add, 100, 20);               // 関数add()を引数にしている
    keisan(sub, 100, 20);
    return 0;
}
                                        // 第1引数が関数ポインタである
void keisan(void calc(int a, int b), int d1, int d2)
{
    calc(d1, d2);                       // 引数calcで渡された関数を実行する
}

void add(int a, int b)
{
    printf("add=%d¥n", a + b);
}

void sub(int a, int b)
{
    printf("sub=%d¥n", a - b);
}
```

実行結果

```
add=120
sub=80
```

第11章

関数

C Quick Reference

078 関数の構成

関数1

Cプログラムは**関数**の寄せ集めとして構成されます。実行文はすべて、どれかの関数にふくまれていなければなりません。

関数には自由に名前をつけることができます。呼び出しが行なわれるたびに、その関数内容が実行されます。

またプログラムの中には、"main"という名前の関数が必要です。プログラムはこのmain()関数から実行開始されます。

関数の一般的な形式は次のとおりです。

関数の形式

```
記憶クラス　戻り値の型　関数名(引数の型と名前)
{
      宣言と文
}
```

記憶クラスはexternとstaticを指定できます。デフォルトはexternであり、通常は省略します。戻り値がない関数の型はvoidを指定します。引数が0個の場合は空のかっこ()または(void)を指定します。

次に典型的な関数を示します。この関数はn1<n2であれば、n1からn2までの総和を返します。

```
   ┌──関数の型はint
   │    ┌──関数名
   │    │     ┌──1番目の引数はint型
   │    │     │          ┌──2番目の引数もint型
int total(int n1, int n2)
{
    int i, sum = 0;
    for (i=n1; i<=n2; i++) {
        sum += i;
    }
    return sum; ──結果を返す
}
```

次にいろいろなスタイルの関数の記述例を示します。

いろいろなスタイルの関数

```
#include <stdio.h>

void msg(void)                  // 戻り値なし 引数なし
{
    printf("半径を入力してください: ");
}

double input(void)              // 戻り値あり 引数なし
{
    double dt;
    scanf("%lf", &dt);
    return dt;
}

double sq_circle(double r)      // 戻り値あり 引数あり
{
    return r * r * 3.14159;
}

void disp(double d)             // 戻り値なし 引数あり
{
    printf("面積=%f¥n", d);
}

int main(void)
{
    double r, s;
    msg();
    r = input();
    s = sq_circle(r);
    disp(s);
    return 0;
}
```

実行結果

```
半径を入力してください: 10.0
面積=314.159000
```

079 関数の記憶クラス

関数2

記憶クラスを指定するexternおよびstaticの機能については「第6章 記憶クラス」(p. 45)で説明していますが、簡単に関数への用法を整理します。

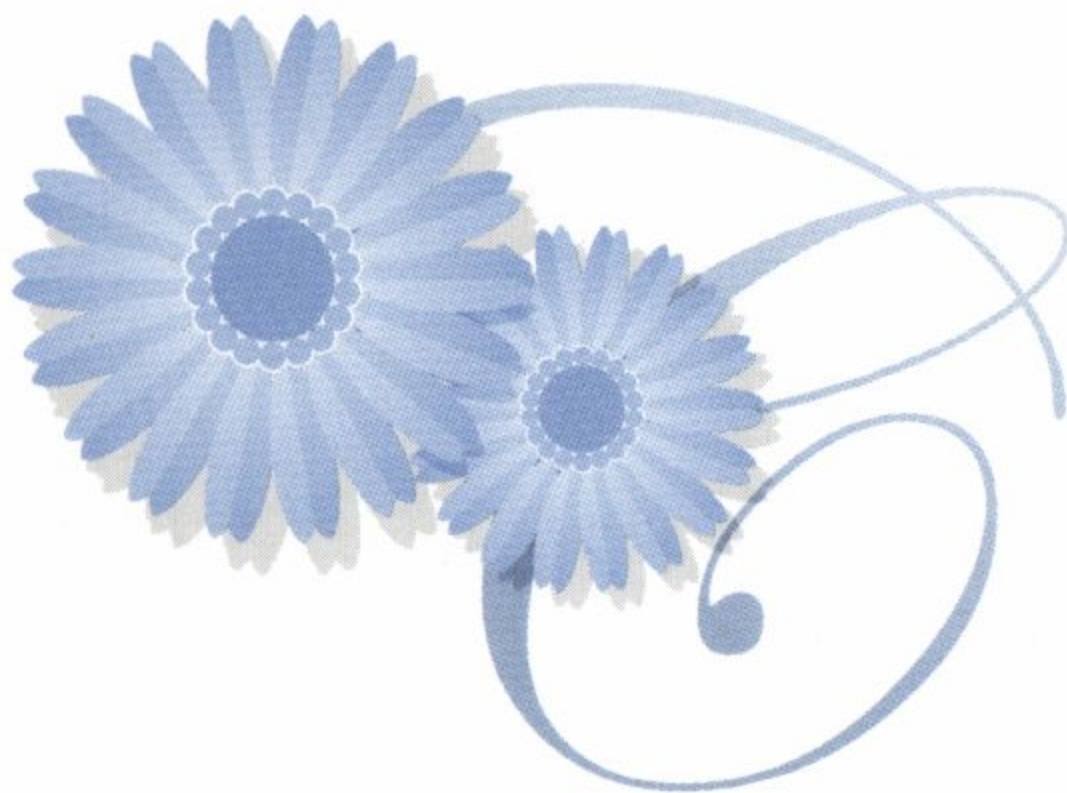
関数の記憶クラス指定子はexternおよびstaticが利用できます。staticのついた関数の情報はリンカに渡されないので、その通用範囲は翻訳単位内に限られます(翻訳単位間で同じ関数名があっても衝突しない)。externのついた関数、もしくはstaticもexternもつかない関数(デフォルトでextern)の情報はリンカに渡され、その関数は他の翻訳単位からも外部参照可能になります。

以下、記述例で説明します。prg1.cとprg2.cは別々の翻訳単位であるものとします。

例1：関数におけるextern指定

```
prg1.c：extern void foo(void) { ... }     ――外部結合の関数
prg2.c：なし                               ――prg1.cのfoo()を利用可能

prg1.c：void foo(void) { ... }            ――同上(デフォルトでextern)
prg2.c：なし                               ――prg1.cのfoo()を利用可能

prg1.c：void foo(void) { ... }            }関数が重複するのでエラー
prg2.c：void foo(void) { ... }            }
```

例2：関数におけるstatic指定

```
prg1.c：static void foo(void) { ... };   }両関数は翻訳単位内のみで通用する
prg2.c：static void foo(void) { ... };   }名前の衝突はない

prg1.c：static void foo(void) { ... };   }他方がstaticつきなので両関数名が
prg2.c：void foo(void) { ... };          }リンク時に衝突することはない
```

080 関数の型とreturn文

関数3

関数の型はその関数がreturn文で戻すデータの型と同じです。関数の型はintやdoubleなど「配列型以外のオブジェクト型、またはvoid型」で指定します。void型は戻り値が不要なときに指定します。

return文に出合うと関数実行は終了し、呼び出し元に制御を戻します。

```
return;      ——void型関数のreturn文
return 10;   ——非void型関数のreturn文と戻り値
```

非void型関数にはreturn文が必須です。void型関数の場合は、return文はなくてもかまいません。記述されていない場合、関数末尾の}に出合うと処理を終了します。

またmain()関数の型はint型ですが、この場合だけはreturn文の記述を省略できます。関数末尾の}に出合うと、C99規格では値0を戻します。それより前のC規格では、戻り値処理は未定義です。

081 戻り値の無視

関数4

Cでは非void型の関数であっても、その戻り値を無視することができます。このとき、本来なら戻るはずの値は単に捨てられます。voidキャストをすれば戻り値無視を明示的にできます。

実例として、たとえば標準関数のstrcpy()も、戻り値をもっていますが、これは無視するのが普通です。

```
ch = getchar();          ——通常の記述
getchar();               ——戻り値無視(一時停止に有用)
(void)getchar();         ——明示的にした
strcpy(s, "abcd");       ——これも戻り値を無視している
```

082 仮引数と実引数

関数5

関数呼び出し側と関数本体側でデータのやりとりをするには引数を用います。引数は関数名に後続するかっこの中に、データ型とともに変数名を書きます。

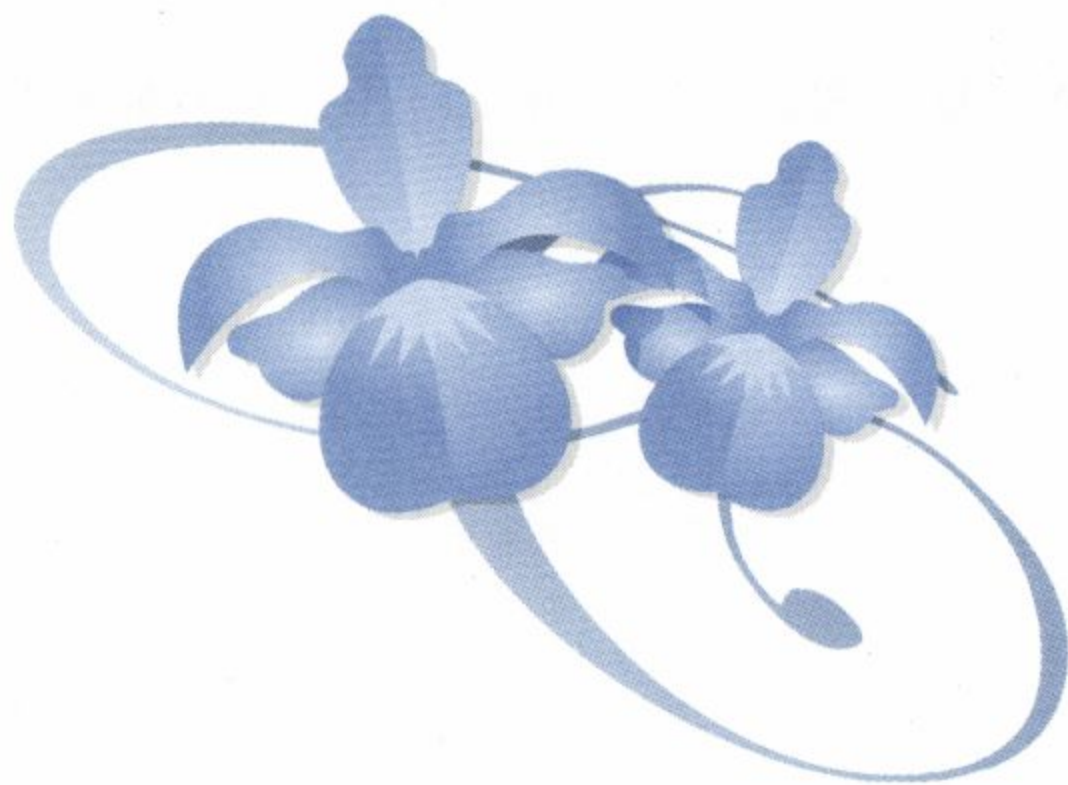
```
int add(int a, int b)  ――int型の変数aとbを引数とする
{
    ...
}
```

この関数定義側の変数aやbを**仮引数**といいます。それはどんな値が渡されるか分からない仮の引数だからです。一方、この関数を呼び出す側では、

```
dt = add(10, 20);      ――実際の値を記述する
dt = add(y, z);        ――実際の変数名を記述する
```

のように実際のデータを記述します。これを**実引数**といいます。

083 可変個の引数

関数6

Cでは**可変個の引数**をとる関数を書くことができます。可変個の引数は次のようにピリオド3個で表現します。

```
int smp(char *form, ...)    ——文字列がひとつと、さらに可変個の引数をとる
{
    処理
}
```

とします。なお「可変個の引数をとる」といっても、固定された最初の引数だけは必要です。この例では仮引数formがそれですが、このformの中に可変個引数を正当に処理するための情報(ヒント)を入れて渡します。実際にその可変個引数を処理する手段は、stdarg.hで提供されています。

この「...」は関数の記述にも、そのプロトタイプ記述にも用います。たとえば可変個引数の代表例であるprintf()関数のプロトタイプは次のようになっています。具体的な可変個引数の例は「160 可変個引数をもつ関数」(p. 241)を参照してください。

```
int printf(const char *format, ...);
```

084 引数の型変換(型の調整) 関数7

関数を呼び出すとき、実引数の型は仮引数の型と一致するように記述します。型が異なっているときは、自動的に型の調整が行なわれます。これについては「関数の引数型変換」(p. 41)に説明があります。次に例を示します。

```
void putdbl(double dt)  ――仮引数はdouble型
{
    printf("%f¥n", dt);
}
...
float f = 12.34F;
int   a = 55;
putdbl(f);              ――float型の引数で呼び出す
putdbl(a);              ――int型の引数で呼び出す
putdbl((double)a);      ――明示的にキャストしてもよい
```

085 配列仮引数とstaticおよび型修飾子 関数8

C99では関数宣言で仮引数に配列を指定するとき、staticおよび型修飾子(const, restrict, volatile)を指定することができます。それは配列の性格を知らせる役目をもちます。次にstaticを使った例を示します。

```
void func1(int dd[static 20]);   ――配列ddが20個以上の要素をもつことを保証する
```

また型修飾子を使うと、通常の「〜型の配列」という意味が、「〜型への○○修飾されたポインタ」という意味になります。

```
void func2(int dd[const]);       ――ddはint型への変更できないポインタ
```

086 可変長配列型の仮引数

関数9

C99では関数の仮引数で、可変長配列であることを示すとき、関数プロトタイプを次のように［*］で記述できます。→「025 可変長配列」(p. 37)

```
void func(int a, int b, int ary[*][*]);   // aryは可変長配列
```

087 関数原型（関数プロトタイプ）

関数10

本体をもつ関数の記述、つまり関数定義が行なわれると、Cコンパイラはそこから関数仕様を収集して管理します。つまり、「関数名と型と引数構成」を登録します。この情報を、

関数原型（関数プロトタイプ）

といいます。あとでその関数が呼ばれたとき、コンパイラは管理している関数原型情報と比較して、正しい関数呼び出し記述であることをチェックします。

このように、関数呼び出しを行なうときは、関数情報が先に判明している必要があります。しかし、記述位置によっては、それができません。次に例を示します。

例1：チェックできる

```
#include <stdio.h>

void foo(int n)         // ここで関数情報が判明している
{
    printf("n=%d¥n", n);
}

int main(void)          // main()関数が後ろにある
{
    foo(10);            // 正しく呼び出すことができる
}
```

例2：チェックできない

```
#include <stdio.h>

int main(void)
{
    foo(10);            // foo()の関数情報がまだ不明
}

void foo(int n)         // ここに関数情報があるが手遅れ
{
    printf("n=%d¥n", n);
}
```

Cコンパイラは、ソースプログラムをワンパスで、先頭から処理していきます。例1では、foo()関数情報が判明してから、それを呼び出しているので問題ありません。例2では、まだ未知のfoo()関数を呼ぼうとしているので、チェックできません。

例2のような関数定義が末尾側にある、あるいは関数定義が別の翻訳単位にある場合に、適切に関数情報を与えるために、関数原型だけを先に宣言する機能があります。次に例を示します。

例3：関数原型の宣言をする

```
#include <stdio.h>
void foo(int n);        // 関数原型を記述

int main(void)
{
    foo(10);            // 関数原型が先にあるのでチェックできる
}

void foo(int n)         // 関数定義
{
    printf("n=%d¥n", n);
}
```

この例で分かるように関数原型は、

――関数定義の先頭行を取り出し、末尾に;をつける

という形式をしています。ただし関数原型の仮引数名は省略できるので、

```
void foo(int);          ――仮引数名は必須ではない
```

としてもかまいません。しかし仮引数名も記述した方が、意味は分かりやすいでしょう。

関数定義は関数原型機能をもっています。そのため関数定義が先に登場するときは、関数原型だけを宣言することは不要です。しかし記述の一貫性のために、すべての関数原型宣言を先に行なうことはよいスタイルです。

088 仮引数情報のない関数宣言

関数11

仮引数情報を入れた関数宣言は関数原型になります。一方、仮引数情報がなく、型指定だけの関数宣言というのもあります。型指定だけの関数宣言は、伝統的なCから引き継ぐ機能です。一方の関数原型はC89規格で導入された新しい機能です。

```
int fnc(int a);    ——(1)関数原型。仮引数が1個
int fnc(void);     ——(2)関数原型。仮引数は0個
int fnc();         ——(3)型指定だけの関数宣言。仮引数情報がないという意味
```

(1)は標準的な関数原型です。(2)は仮引数がないことを示す関数原型です。(3)は「仮引数情報のない関数宣言」です。この記述は仮引数がないという意味ではなく、

——仮引数の情報がない(仮引数構成についてはチェックしない)

という意味になります。仮引数の情報がないので、Cコンパイラは関数呼び出し時に関数型のみをチェックし、引数構成のチェックは行ないません。この空のかっこをともなう関数宣言は、チェック能力が低く、現在のC規格では今後の言語の方針として「廃止予定事項」にされています。

次のプログラムは仮引数情報がありませんが、コンパイル自体は正しく行なわれます。問題のあるプログラムです。

```
#include <stdio.h>
void foo();             // 型情報のみで、仮引数情報はない
int main(void)
{
    foo();              // 不正だがコンパイルOK
    foo(10);            // 正しい
    foo(10, 20);        // 不正だがコンパイルOK
}

void foo(int n)
{
    printf("n=%d¥n", n);
}
```

089 伝統的Cの関数定義スタイル 関数12

伝統的Cでは、関数定義において、仮引数名と、その型指定を別べつに指定していました。このスタイルは現在でも利用可能ですが、今後の言語の方針として「廃止予定事項」になっています。この定義スタイルは関数原型としては働きません。適切にチェックさせるには、関数原型の記述が必要になります。

```
#include <stdio.h>

void foo(x, y)              // 仮引数名と型指定を別べつに記述するスタイル
double x;
int y;
{
    printf("x=%f y=%d¥n", x, y);
}

int main(void)
{
    foo(12.34, 55);          // 正しい
    foo(12.34, 55, 66);      // 不正だがコンパイルOK
}
```

090 暗黙の関数型宣言

関数13

適切な関数原型情報があると、戻り値型と引数構成がチェックされます。また引数情報のない関数宣言があると、戻り値の型のみチェックします。どちらもない場合、関数の戻り値の型に関して「int型であると仮定する」というルールで処理されます。つまり**暗黙の関数型宣言**が行なわれます。

関数型情報がないプログラムをコンパイルすると、警告が表示されるかもしれませんが、その時点での処理は正しく行なわれます。

ただし、あとで関数定義が登場し、その関数型がintではなかった場合は、仮定したint型と異なるので、再定義エラーになります。このような暗黙の型宣言に頼るプログラムは安全性の面で望ましくありません。関数原型を正しく書くのがよいスタイルです。

次に、関数原型がないプログラムの警告例を示します。

プログラム例

```
#include <stdio.h>
int main(void)
{
    printf("%d¥n", intf(10));
    printf("%f¥n", dblf(2.34));
    return 0;
}

int intf(int a) { return a * a; }
double dblf(double a) { return a * a; }
```

コンパイル時の警告例

```
t1.c(4) : warning C4013: 関数 'intf' は定義されていません。int型の値を返す外部関数と見なします。
t1.c(5) : warning C4013: 関数 'dblf' は定義されていません。int型の値を返す外部関数と見なします。
t1.c(11) : error C2371: 'dblf' : 再定義されています。異なる基本型です。
```

091 標準ライブラリ関数の関数原型指定 関数14

標準関数も関数原型を必要とするのは同じです。標準関数の関数原型は、ヘッダファイルにまとめられているので、それをinclude指定します。たとえばstrcpy()関数の関数原型はstring.hに入っているので、それを指定します。標準関数が必要とするヘッダファイルは、リファレンスマニュアル情報として提供されます。

```
#include <string.h>  ——これを書くと
strcpy(s,"abcd");    ——この関数呼び出しがチェックされる
```

092 データを渡す方法1 関数15

Cでは関数への引数渡しは「値による呼び出し(Call by value)」で行なわれます。

またこのメカニズムを利用してアドレス(という値)を渡せば、「アドレスによる呼び出し」を行なうことができます。このとき呼び出された関数側では、そのアドレスにある値を参照することで、値を利用できますし、呼び出し側オブジェクトの内容を変更することができます。ここではいろいろなデータ渡しの方法を説明します。

値を渡す

通常のオブジェクト値を渡すことができます。呼び出し関数側の実引数は、関数定義側の仮引数にコピーされます。データを渡し終えると、実引数と仮引数は無関係になりますから、関数側で操作した仮引数値が、呼び出し側のオブジェクトに影響を与えることはありません。関数の独立性や安全性の面で有益です。

```
void disp(int n)      // 値を受け取る
{
    printf("%d¥n", n);
}

void test(void)
{
    int a = 10;
    disp(a);          // 出力：10
    disp(20);         // 出力：20
}
```

アドレスを渡す

オブジェクトのアドレス（ポインタ）を渡すことができます。アドレスを介することで、呼び出し関数側と呼ばれる関数側とで、同一のオブジェクトを参照できます。この機能を使うと、戻り値を使わずに、呼び出し側に処理結果を返す効果を得ることもできます。

次の例でout()関数はオブジェクトのアドレスを受け取り、間接参照して内容を表示しています。swap()関数はアドレスを介して、呼び出し側のオブジェクトの内容を交換しています。

```
void out(int *n)                    // 引数をポインタにする
{
    printf("%d¥n", *n);             // 間接参照 *n で内容を表示
}

void swap(int *a, int *b)           // ふたつのポインタを受け取る
{
    int t;
    t = *a; *a = *b; *b = t;        // オブジェクト内容交換
}

void test(void)
{
    int a = 10, b = 20;
    out(&a);                        // 出力：10
    out(&b);                        // 出力：20
    swap(&a, &b);                   // スワップする
    printf("%d %d¥n", a, b);        // 出力：20 10
}
```

093 データを渡す方法2

関数16

void型ポインタを仮引数にする

関数仮引数としてvoid型ポインタを指定することができます。void型ポインタを使うと、

――どのような型のオブジェクトのアドレスも受け取ることができる

という技巧的な活用ができます。関数の多重定義ができないC言語では貴重な機能です。

次のv_disp()関数は第1引数でデータ種別を指定して、void型ポインタの切り分け処理を行ないます。データは適切にキャストして取り出しています。

```
void v_disp(int typ, void *p)
{
    switch (typ) {
    case 'i': printf("%d¥n", *(int *)p);    break;
    case 'd': printf("%f¥n", *(double *)p); break;
    case 's': printf("%s¥n", (char *)p);    break;
    default:  printf("?¥n");
    }
}

void test(void)
{
    int idt = 112233;
    double ddt = 456.789;
    char sdt[10] = "abcdef";
    v_disp('i', &idt);      // 出力：112233
    v_disp('d', &ddt);      // 出力：456.789000
    v_disp('s', sdt);       // 出力：abcdef
}
```

配列を渡す

配列を渡す場合は、その先頭アドレスを渡します。このとき仮引数は、

(1)ポインタにする
(2)配列形式にする

というふたつの指定方法があります。配列を渡す場合、その先頭アドレスを伝えるだけで、配列の長さ情報を伝えることはできません。長さ情報はプログラマが技巧的に処理します。

二次元以上の配列を渡すときは配列の構造を伝える必要があります。そのため2次元目以降の配列サイズを明示します。

なお仮引数の配列サイズ(多次元配列の場合は先頭次元のサイズ)は、記述してもエラーにはなりませんが無視されます。たとえば次の例でvoid sum2(int d[10])と記述してもvoid sum2(int d[])とみなされます。

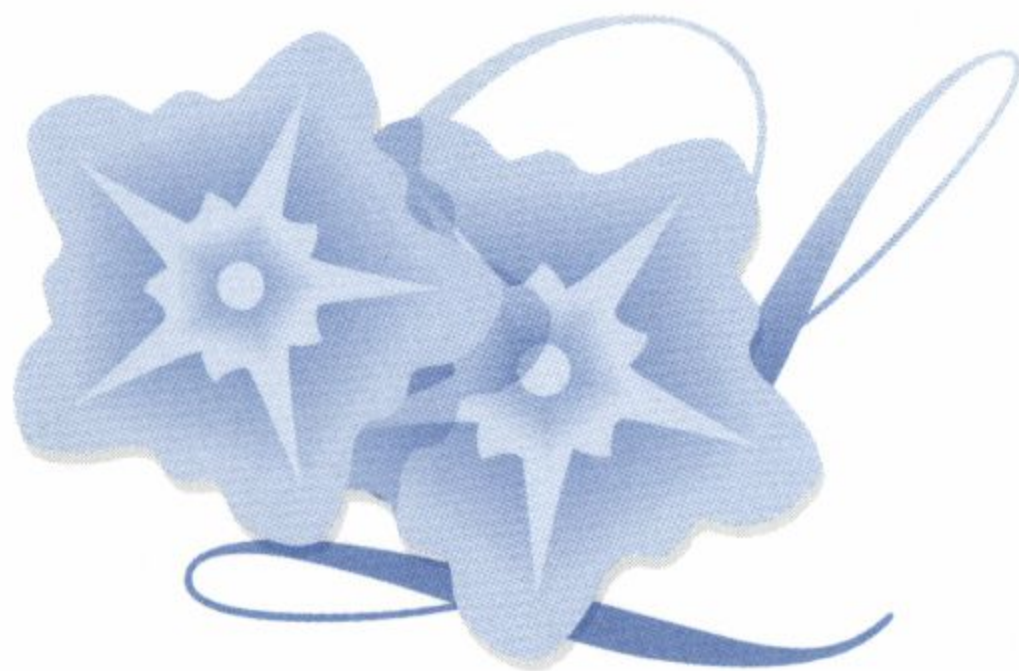
```
void sum1(int *p)                          // 仮引数をポインタ表現する
{
    int sum = 0;
    while (*p != -1)
        sum += *p++;                       // 合計値を計算
    printf("sum=%d¥n", sum);
}

void sum2(int d[])                         // 仮引数を配列表現する
{
    int i, sum = 0;
    for (i=0; d[i] != -1; i++)             // 合計値を計算
        sum += d[i];
    printf("sum=%d¥n", sum);
}

void disp(int d[][4], int n)               // 2次元目サイズは4
{
    int i, j;
    for (i=0; i<n; i++) {                  // nで1次元目サイズを知る
        for (j=0; j<4; j++) { printf("%d ", d[i][j]); }
        putchar('¥n');
    }
}

void test(void)
{
    int a[6] = {1, 4, 6, 2, 8, -1};        // -1は終端マーク
    int b[2][4] = {{10, 11, 12, 13},{20, 21, 22, 23}};
    sum1(a);                               // 出力：sum=21
    sum2(a);                               // 出力：sum=21
    disp(b, 2);                            // 出力：10 11 12 13
}                                          //       20 21 22 23
```

文字列を渡す

文字列を関数に渡す場合は、すでに説明した配列渡しの方法を使うだけです。ポインタ形式で、あるいは配列形式で仮引数を記述します。文字列引数の場合は'¥0'で末尾を知ることができるので便利です。次のプログラムは、渡された文字列の大文字小文字を逆にしています。

```
void change(char *p)    // 仮引数は(char p[])でも同じ
{
    while (*p) {
        if (isupper(*p))        *p = tolower(*p);
        else if (islower(*p))   *p = toupper(*p);
        ++p;
    }
}

void test(void)
{
    char s[] = "abc123XYZ";
    change(s);
    puts(s);                // 出力：ABC123xyz
}
```

094 戻り値を返す方法

関数17

値を返す

関数から返す値はreturn文で記述します。次のプログラムは大きい方の値を返す関数maxdt()を記述したものです。

```
int maxdt(int a, int b)
{
    if (a > b)          // a>bなら
        return a;       // aを返す
    else                // そうでなければ
        return b;       // bを返す
}
```

ポインタを返す

ポインタを返す場合は、そのポインタの示すオブジェクトがどこにあるかに注意します。

引数として渡されたアドレスを返すときは記述は簡単です。

また関数内にオブジェクトを確保する場合は、関数が終了しても有効であるように、静的変数にしておきます。

次のmax_str()関数は、大きい方の文字列のポインタを返します。戻り値ポインタは、渡されたポインタのどちらかです。to_upstr()関数は、渡された文字列を大文字変換して、関数内の静的配列に保存し、その先頭アドレスを返します。

```
char *max_str(char *s1, char *s2)
{
    if (strcmp(s1, s2) >= 0) return s1;   // s1>=s2ならs1を返す
    return s2;                            // そうでないならs2を返す
}

char *to_upstr(char *s)
{
    int n;
    static char buf[101];                 // 変換後文字列をここに格納

    for (n=0; n<100; n++)                 // 最大100文字
        buf[n] = toupper(*s++);           // 大文字にして格納
    buf[n] = '¥0';
    return buf;                           // 先頭を返す
}
```

```
void test(void)
{
    char *p, a[] = "abcd", b[] = "efgh";
    p = max_str(a, b);
    puts(p);                            // 出力：efgh
    p = to_upstr(a);
    printf("p=%s a=%s¥n", p, a);        // 出力：p=ABCD a=abcd
}
```

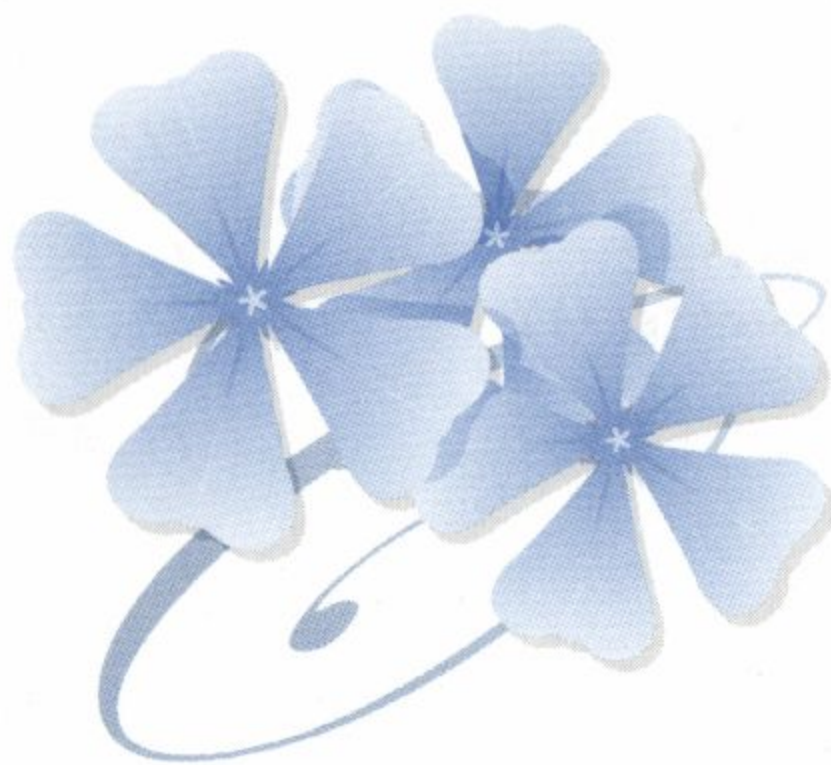
複数の戻り値

return文では複数個の戻り値を返すことはできません。引数をポインタにして、呼び出し側と関数側でデータを共有すると、見かけ上、複数のデータを返したようにすることができます。

次のプログラムはdouble値を渡すと、その整数部と小数部を返します。

```
void int_deci(double dt, int *intdt, double *decidt)
{
    *intdt = (int)dt;
    *decidt = dt - (int)dt;
}

void test(void)
{
    int a;
    double b, dt = 12.345;
    int_deci(dt, &a, &b);
    printf("%f %d %f¥n", dt, a, b);    // 出力：12.345000 12 0.345000
}
```

095 インライン関数

関数18

インライン関数は、C99で追加されたもので、形式は関数ですが、コード自身を記述位置に埋め込むという処理を行ないます。関数呼び出しを行なわないので、関数形式マクロを使った場合と同じように、処理が速くなります。引数は関数ルールで処理されます。マクロ定義のような引数の複数回評価による不都合が生じることはありません。

ただし規格では「その関数の呼び出しを可能な限り速くすることを示唆する。その示唆が効果をもつ程度は処理系定義とする」となっています。したがって処理系は、コード埋め込みではない方法を使うこともできるし、inlineという記述を単に無視することもできます。その場合は通常の関数として働きます。

```
inline int func(int n)      // inline指定子があるとインライン関数になる
{
    ...
}
```

プログラム例：インライン関数の機能を確認する

```
#include <stdio.h>
inline int sub(int a, int b)      // inlineを指定
{
    return a * b;
}

int main(void)
{
    int c;
    c = sub(10, 20);
    printf("%d¥n", c);
    return 0;
}
```

実行結果

注：GNU Cでコンパイルオプション「--save-temps -O」を指定すると、コンパイル結果のアセンブラファイルmyprog.sが生成される。その内容は以下のようになる。

```
main:
pushl   %ebp
movl    %esp, %ebp
andl    $-16, %esp
subl    $16, %esp
movl    $200, 4(%esp) ―― sub()関数がインライン展開され最適化で定数200になった
movl    $.LC0, (%esp)
call    printf
movl    $0, %eax
leave
ret
```

096 main関数の処理

関数19

main関数の実行

Cプログラムはmainという名前の関数が必要です。Cプログラムはmain()関数から実行開始します。

C規格ではmain()関数は次のどちらかの定義を行ないます。仮引数名は自由ですが、argcとargvを用いるのが慣例です。

```
int main(void) { ~ }                       ——(1)引数をもたない場合
int main(int argc, char *argv[]) { ~ }     ——(2)引数をもつ場合
```

main関数の終了

プログラムはmain()関数の終端の'}'に出合うと終了します。またmain()関数内に「return 値;」があるとその位置で終了します。このとき「値」は、このプログラムを呼び出したOS側へ戻す終了コードとなります。

main()関数内にreturn文がないときは、末尾の}に出合うと終了します。このときの終了コードはC99では0、それより前のC規格では未定義です。

OSへの終了コードは次のexit()関数の仮引数で用いるものと同じです。正常終了なら0、異常終了なら1以上とするのが慣例です。

このreturn文でプログラム終了するのは、main()関数の中に記述したときだけです。return文は「呼び出し側に制御を戻す」という役目をもちますが、main()関数はOSから呼ばれているので、OSに戻る(=プログラムを終了する)ことになります。

なおCプログラムは、exit()関数に出合うことでも終了します。exit()は一般の関数の内部に記述できます。

main関数への引数渡し

上の(2)の構文を使うと、コマンドラインから入力した文字列を、main()関数に引数として渡すことができます。main()関数へ渡すことのできる情報は、基本的には、

引数の総個数
引数の文字列実体(を指すポインタ)

のふたつです。このとき各引数には、

argc	――引数の数
argv[0]	――0番目（最初）の引数文字列（=コマンド名）
argv[1]	――1番目の引数文字列
argv[2]	――2番目の引数文字列
⋮	⋮

という形でデータが渡されます。

次にmain()関数の引数処理を確認するプログラム例を示します。

main関数への引数渡しの確認

```
/* argtst.c */
#include <stdio.h>

int main(int argc, char *argv[])
{
    int n;
    for (n=0; n<argc; n++) {
        printf("n=%d: %s¥n", n, argv[n]);
    }
    return 0;
}
```

実行結果：Visual C++を使い、DOS窓で実行

```
C:¥cwork>argtst aaa bbb /w↵
n=0: argtst
n=1: aaa
n=2: bbb
n=3: /w
```

第12章

構造体型

C Quick Reference

097 構造体の宣言と参照

構造体型１

構造体の宣言

構造体は既存のデータ型を組み合わせて、新しいデータ型を作ります。構造体は複数のデータ型をひとつの名のもとに組織化するという働きをします。

構造体は次のように記述します。**構造体タグ**は構造体の名前です。また構造体要素である変数のことを**メンバ**といいます。

構造体の宣言

```
struct  構造体タグopt { メンバ宣言並び }
```

次に簡単な構造体の例を示します。

```
struct intdbl {        // 構造体名をintdblにする
    int i;
    double d;
};

struct intdbl dt;      // 構造体intdbl型の変数dt
```

メンバの参照

構造体のメンバ参照は次のように行ないます。構造体の変数名から参照する場合と、ポインタを介して参照する場合の、ふたつの方法があります。

```
struct intdbl dt;
struct intdbl *pt = &dt;
...
dt.i = 100;     // 変数のときは . でアクセス
pt->i = 100;    // ポインタのときは -> でアクセス
```

ポインタの場合も . で指定できますが、間接参照が必要なので、「(*pt).i = 100;」と書くことになります。便宜のために専用のアクセス演算子 -> が用意されています。

構造体型の変数の宣言方法

構造体型の変数の宣言には、いろいろな方法があります。構造体名を省略することもできます。いろいろな宣言の方法を次に整理します。

■①構造体を宣言し、変数を宣言する（すでに説明）

```
struct intdbl {          // まずintdbl型を記述し
    int i;
    double d;
};

struct intdbl dt;        // intdbl型の変数を宣言
```

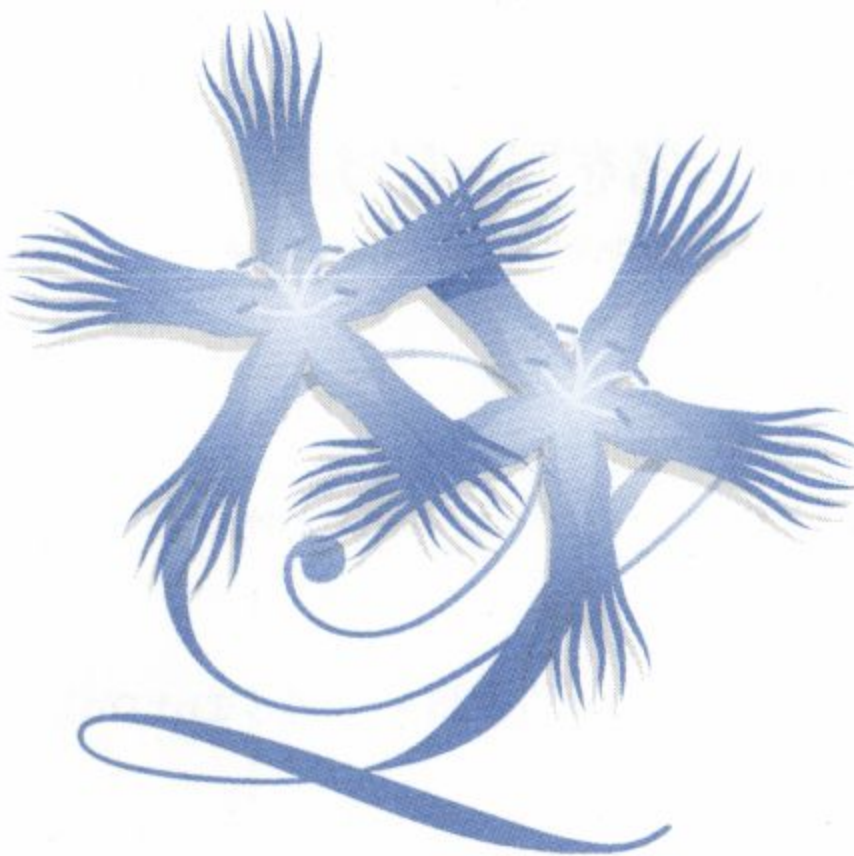
■②上記ふたつを同時に記述する

```
struct intdbl {          // タグ名と
    int i;
    double d;
} dt;                    // 変数を同時に指定する
```

変数名は「struct intdbl { ... } a, b, c;」のように複数指定できる。

■③タグ名の省略

タグ名を二度と使わないなら、これを省略していきなり「タグ名はないが、ある構造をもつ変数」を宣言できる。

```
struct {                 // タグ名はないがこの構造をもつ
    int i;
    double d;
} dt;                    // 変数dtを宣言
```

098 構造体宣言とtypedefの併用 構造体型2

構造体型の変数宣言では、「struct intdbl dt;」のようにstructを先行させる必要があります。typedefを使って同義語を作ると、これが不要になります。いろいろな記述例を示します。

①構造体宣言後にtypedefで別名宣言

```
struct intdbl {                      // 構造体intdblの宣言
    ...
};

typedef struct intdbl IntDbl;        // 同義語指定
IntDbl dt;                           // struct intdbl dt;と解釈される
struct intdbl dt;                    // 通常の記述もOK
```

②typedef宣言と構造体宣言を一体化する

```
typedef struct intdbl {              // typedefつき構造体宣言
    int i;
    double d;
} IntDbl;                            // ここで同義語指定

IntDbl dt;
```

typedef宣言名は「struct intdbl { ... } Name1, Name2;」のように複数指定できる。

typedefの有無で}に後続する識別子の意味が異なるので注意。

```
struct intdbl { ... } dt;                 // dtは変数名
typedef struct intdbl { ... } IntDbl;     // IntDblはtypedef対応名
```

③構造体名とtypedef宣言名を同一にする

文法的に構造体名とtypedef宣言名は別空間なので同一名でもよい(推奨はできない)。

```
typedef struct IntDbl {    // 構造体名はIntDbl
    int i;
    double d;
} IntDbl;                  // 同義語名も同じIntDbl
```

■④構造体名を使わない

構造体名を使うことがないなら省略してtypedef宣言名だけにすることもできる。

```
typedef struct {        // 構造体名がない
    int i;
    double d;
} IntDbl;               // 同義語名をつける

IntDbl dt;              // このように記述できる
```

099 構造体の初期化

構造体型3

構造体は初期化できます。初期化のルールは、一般オブジェクトの初期化と同様ですが、初期値は{}とともに記述します。初期化要素が不足しているときは、残りの要素は0になります。初期化は定数式で行ないます。C99では自動記憶域期間をもつ構造体は「式」で初期化できます。

次に初期化の例を示します。

①初期化例1：単独の初期化

```
struct typSN {
    char str[40];
    int  nbr;
};

struct typSN a = { "hello", 123 };      // 初期化をする
struct typSN b = { "hello" };           // {"hello",0} と同じ
struct typSN c = a;                     // aの内容で設定
```

②初期化例2：構造体宣言時の初期化

```
struct typSN {
    char str[40];
    int  nbr;
} a = { "hello", 123 };
```

③初期化例3：配列の初期化

```
struct typSN n[3] = {  // 添字は省略もできる
    { "aaaa", 100 },   //┐添字を省略するとこの数を数えて
    { "bbbb", 200 },   //│合理的な配列長を設定してくれる
    { "cccc", 300 }    //┘この場合3とみなす
};

struct typSN n[3] = {
    "aaaa", 100,       //┐初期値の対応が正しいなら
    "bbbb", 200,       //│内側の{}がなくてもOK
    "cccc", 300        //┘
};
```

④初期化例4：C99では特定のメンバを指定して初期化することができる

「042 要素指示子を使う初期化」(p. 65) 参照。

```
struct typM {
    int a, b, c, d, e;
};
struct typM dd = { .b=11, .d=22 }; // 内容 → { 0, 11, 0, 22, 0 }
```

100 構造体の演算

構造体型4

構造体に対する演算は次のものだけが用意されています。通常のデータ操作は、個別にメンバを指定して行ないます。

■①構造体のコピー

構造体を丸ごとコピーすることができます。構造体の中に文字列があっても、そのままコピーしてくれます。

この機能があるため、関数への引数渡しや、関数からの戻り値として、構造体を用いることができます。

```
struct intdbl d1, d2, dd[10];

d1 = d2;            // 構造体のコピー
dd[2] = dd[5];      // 配列内容のコピー
```

■②構造体のアドレスを取り出す

これは構造体をポインタで操作するときに用います。また関数に構造体をアドレス渡しするときにも用います。

```
struct intdbl dt, *pt;
pt = &dt;           // dtのアドレスを取り出して代入
```

■③メンバを参照する

すでに説明したとおりメンバアクセス演算子を使うことで構造体のメンバを個別に指定することができます。

```
struct intdbl dt, *pt = &dt;

dt.i = 100;         // オブジェクトの場合
pt->i = 100;        // ポインタの場合
```

101 構造体の関数との受け渡し

構造体型5

構造体は通常の変数と同じように、構造体そのもの、あるいはポインタを引数や戻り値にすることができます。

次に簡単な利用例を示します。add()関数はふたつの構造体値をメンバごとに加算し、値を返します。maxdt()関数はポインタ渡しされた構造体内容を面積として比較し、大きい方のポインタを返します。

構造体を引数と戻り値にするプログラム

```
#include <stdio.h>
struct XYdata { int x, y; };
struct XYdata add(struct XYdata a, struct XYdata b);
struct XYdata *maxdt(struct XYdata *a, struct XYdata *b);

int main(void)
{
    struct XYdata d1 = {10, 20}, d2 = {40, 60}, d3, *pt;
    pt = maxdt(&d1, &d2);                    // ポインタ処理
    printf("%d %d¥n", pt->x, pt->y);         // 出力：40 60
    d3 = add(d1, d2);                        // 値処理
    printf("%d %d¥n", d3.x, d3.y);           // 出力：50 80
    return 0;
}

struct XYdata add(struct XYdata a, struct XYdata b)          // 加算
{
    a.x += b.x;
    a.y += b.y;
    return a;
}

struct XYdata *maxdt(struct XYdata *a, struct XYdata *b)    // MAX値
{
    if (a->x * a->y >= b->x * b->y) return a; else return b;
}
```

102 構造体タグの宣言

構造体型6

構造体を使う場合は、まず構造体を宣言し、そのあとで、その構造体を使う関数を記述するのが基本です。

しかし構造体宣言は位置の取り合いが起きることがあります。このタイミングの問題をサポートするため、識別子（構造体タグ）だけを先に宣言しておく機能が用意されています。次の例は、仮宣言があるので、関数プロトタイプが警告なしになります。

構造体タグだけを先に宣言する例

```
struct xydata;                  // 仮宣言を先に書く
void foo(struct xydata a);      // この関数プロトタイプは無警告になる

struct xydata {                 // 構造体本体はあとで登場
    int x,y;
};
```

103 入れ子の構造体

構造体型7

構造体はそのメンバに別の構造体をふくむことができます。構造体にふくまれる構造体のメンバを指定するときは、アクセス演算子を多重に使用します。

簡単な記述例を示します。

```
struct Dset {               // 構造体Dsetを先に記述
    double d1, d2;
};

struct Iset {
    int i1, i2;
    struct Dset d;          // 構造体Dsetをふくむ
};

struct Iset n;              // Iset型の変数
n.i1 = 100;                 // Isetのメンバi1への値設定
n.d.d1 = 33.44;             // Isetのメンバdのメンバd1への値設定
```

104 自己参照構造体

構造体型8

入れ子の構造体では、構造体のメンバとして別の構造体を設定できましたが、これとは別に、構造体のメンバとして、「自分自身へのポインタ」を設定することができます。これを**自己参照構造体**といいます。自己参照構造体はリスト構造を実現するときに便利です。

次のプログラムは単純なリスト構造モデルを記述したものです。3組のデータを自己参照構造体で実現したリストにつないでいきます。最後にリストをたどりながら、全内容を表示します。

ここでは単純な末尾追加を行なっていますが、先頭、途中、末尾のデータの挿入・削除処理などが、リストチェーンをつなぎ替えることで簡単に実現できます。

自己参照構造体の利用例

```
#include <stdio.h>
#include <stdlib.h> /* for atoi() exit() malloc() */
#include <string.h> /* for strcpy() */

typedef struct person {                // 構造体の宣言
    char name[80];
    int age;
    struct person *next;               // 自己参照用のメンバ
} Person;
Person *start = NULL;                  // まだデータは0件

void set_Person(char *nm, int ag)  // リストに追加
{
    Person *tmp, *wp;

    /* Person 1個分のメモリ確保 */
    tmp = (Person *)malloc(sizeof(Person));
    if (tmp == NULL) { printf("メモリ確保できません.¥n"); exit(1); }

    /* 値設定 */
    strcpy(tmp->name, nm);
    tmp->age = ag;
    tmp->next = NULL;

    /* 末尾に追加設定する */
    if (start == NULL) { start = tmp; return; }
    for (wp=start; ; wp = wp->next) {
        if (wp->next == NULL)  { wp->next = tmp; break; }
    }
}
```

```
void disp_Person(void)                 // リスト内容表示
{
    Person *wp;
    for (wp=start; wp!=NULL; wp=wp->next) {
        printf("name:%s age:%d¥n", wp->name, wp->age);
    }
}

int main(void)
{
    set_Person("Tanaka", 40);
    set_Person("Yamada", 20);
    set_Person("Suzuki", 30);
    disp_Person();
    return 0;
}
```

実行結果

```
name:Tanaka age:40
name:Yamada age:20
name:Suzuki age:30
```

上に示したたリスト構造は後方へたどることのできる、片方向チェーンになっています。もし前方向にもたどることのできる、双方向チェーンにしたいときは、次の形式の自己参照構造体にします。

```
typedef struct person {         // 構造体の宣言
    char name[80];
    int age;
    struct person *before;      // 前へのチェーン
    struct person *next;        // 次へのチェーン
} Person;
```

105 フレキシブル配列メンバ

構造体型9

C99では構造体がふたつ以上の名前つきメンバをもつ場合、最後のメンバは、不完全配列型であってもかまいません。これを**フレキシブル配列メンバ**といいます。「フレキシブル配列メンバ」をもつ構造体は、実行時に最後のメンバのサイズを与えて使用します。

定義例

```
struct Styp {
    int n;
    double d[];  // フレキシブル配列メンバ
};
```

使用例：d[]のサイズを10にする

```
struct Styp *st;
st = malloc(sizeof(struct Styp) + sizeof(double)*10);
st->d[5] = 123.456;
```

このときstの指す構造体は次のように定義されているように働きます。ただしこの時点でもsizeof(struct Styp)は4のままです。

```
struct Styp { int n;  double d[10]; }
```

第13章

ビットフィールド

C Quick Reference

106 ビットフィールドの宣言 ビットフィールド1

構造体のメンバは次の形式で、値表現のためのビット幅を指定することができます。このようなメンバを**ビットフィールド**(**bit field**)といいます。

ビットフィールドの指定

```
データ型　メンバ名 : ビット幅;
```

ビットフィールドに用いるデータ型はunsigned int, int, signed intが許されますが、通常はunsigned int型が用いられます。

次に簡単なビットフィールドの用法を示します。メンバaとbは1ビット幅表現なので、0/1の値を保存できます。メンバcは3ビット幅表現なので、0〜7の値を保存できます。

```
struct Bittyp {
    unsigned int a: 1;
    unsigned int b: 1;
    unsigned int c: 3;
};

void test(void)
{
    struct Bittyp dt;
    dt.a = 0;
    dt.b = 1;
    dt.c = 7;
    printf("%d %d %d¥n", dt.a, dt.b, dt.c);     // 出力:0 1 7
}
```

107 ビットフィールドの初期化 ビットフィールド2

ビットフィールドは標準の構造体と同じように初期化できます。次に例を示します。

```
struct Bittyp dt = { 0, 1, 7};    ――初期化する
```

108 無名のビットフィールド ビットフィールド3

名前のないビットフィールドを指定できます。これはビット確保位置を調整する詰め物(非使用ビット)として使われます。これは仕様書などでビット配列がはじめから指定されているときの記述に便利です。

```
struct bitset {
    unsigned int busy  : 1;
    unsigned int ready : 1;
    unsigned int       : 2;    ――2ビットの詰め物
    unsigned int send  : 1;
    unsigned int rec   : 1;
    unsigned int mode  : 3;
};
```

規則では「先頭に名前のない詰め物があってはならない」ことになっています。その場合は「unsigned int no_use : 1;」のようにして、この名前を使わないという記述で対応できます。

109 無名で幅0のビットフィールド ビットフィールド4

ビットフィールドは、「それを保持するに十分な大きさの任意のアドレス付け可能な記憶単位に割り付け」されることになっています（以下「任意の記憶単位」と略称する）。具体的にどのように割り付けるかは処理系依存です。ここでは「任意の記憶単位」を32ビットワードとして説明します。

ビットフィールドで指定したビット数が「任意の記憶単位」に収まりきれないときは、残ったビット数は隣接した次の「任意の記憶単位」に格納されます。その場合、複数ビットを所有するメンバ、たとえば、

```
unsigned int dt: 6;  ――6ビットで構成されるメンバ
```

が境界部分にあるとき、

――前の「任意の記憶単位」にnビット、次の「任意の記憶単位」に6-nビット

のように分断させるか（語境界をまたぐか）、あるいは全ビットをそっくり次の「任意の記憶単位」に送るかという処理方法があり、どちらをとるかは処理系依存です。

この語境界は、プログラマが強制的に指定することができます。それには、「無名でビット幅0」という特別のメンバを使います。

```
struct bitset {
    unsigned int a   : 10;  } 最初の記憶単位に格納される
    unsigned int b   : 12   }
    unsigned int     : 0;   ――これで強制的に語境界を作る
    unsigned int c   : 6;   } 次の記憶単位に格納される
    unsigned int d   : 2;   }
};
```

110 ビットフィールドの混在 ビットフィールド5

ビットフィールドは構造体のメンバですから、通常のメンバ記述と混在させてもかまいません。次に例を示します。

```
struct model {
    int  idt;                    // 通常のメンバ
    double ddt;                  // 通常のメンバ
    unsigned int bit1 : 1;       // 以降ビットフィールド
    unsigned int bit2 : 1;
    unsigned int bit3 : 1;
};
```

第14章 共用体

C Quick Reference

111 共用体の宣言

共用体1

共用体はキーワードunionを使い、構造体と同様の形式で、新しいデータ型を作ります。共用体のすべてのメンバは、同じアドレスからはじまる記憶域に配置されます。つまり同じ記憶域を共用します。

共用体を使うと、同じ記憶域の内容を、異なる名前で参照したり、異なる型として参照したりできます。

また構造体の場合と同じように、共用体の中にもビットフィールドを記述することができます。

共用体の宣言

union 共用体タグ$_{opt}$ **{** メンバ宣言並び **}**

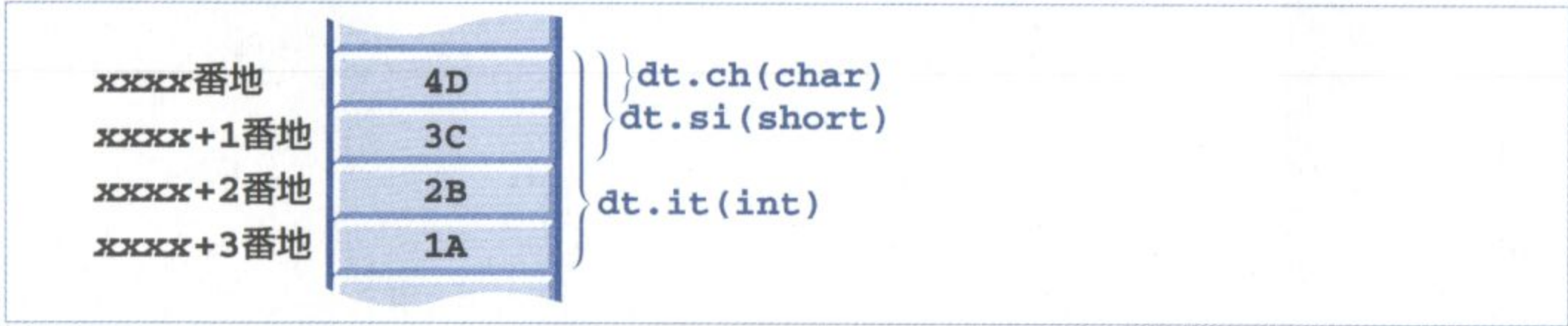
次に簡単な共用体の例を示します。

```
union Utyp {                              // 共用体を宣言
    int it;
    short int si;
    char ch;
};

void test(void)
{
    union Utyp dt;
    dt.it = 0x1A2B3C4D;
    printf("%X %X %X¥n", dt.it, dt.si, dt.ch);
                                          // 出力：1A2B3C4D 3C4D 4D
}
```

この出力例はインテル系CPU上で実行したものです。インテル系CPUでは格納数値のバイト配置はリトルエンディアンと呼ばれる逆並び（LSBからMSBへ）になります。この共用体のメモリ配置例を示します。

共用体のメモリ配置例

112 共用体の初期化

共用体2

共用体の初期化は、その最初のメンバに対して行ないます。初期値は{}で囲みます。ただしC99では「名前つきメンバ」の指定をすることで、任意のメンバを初期化できるようになりました。

初期化は定数式で行ないます。C99では自動記憶域期間をもつ共用体は「式」で初期化できます。

前項のUtyp型を使った初期化の例を示します。

```
union Utyp dt = { 1000 };           // dt.itを初期化する
union Utyp dt = { .si = 1000 };     // C99ではこの記述も可
```

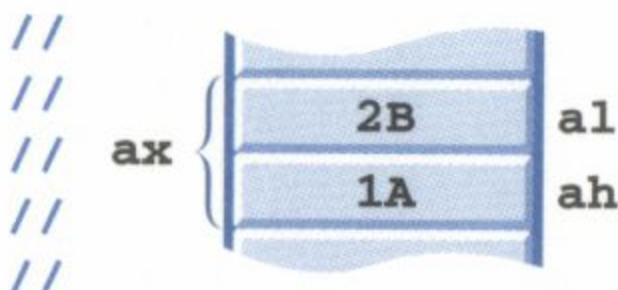
113 共用体と構造体の組み合わせ

共用体3

共用体と構造体は組み合わせて使用できます。例としてレジスタ構成をモデル化することを考えてみましょう。

8086系MPUでは16ビットレジスタであるaxの上位バイト、下位バイトをそれぞれah, alという名前で操作することができます。この構造体は次のように実現できます。

共用体で86系レジスタを表現する

```
#include <stdio.h>

struct hregs {
    unsigned char al;
    unsigned char ah;
};

typedef union regs {         //
    struct hregs h;          //
    unsigned short ax;       //
} Regs;                      //
                             //
int main(void)
{
    Regs rg;
    rg.h.ah = 0x1A;
    rg.h.al = 0x2B;
    printf("%X %X %X¥n", rg.h.ah, rg.h.al, rg.ax);
                                            // 出力：1A 2B 1A2B
    return 0;
}
```


なお本格的にこのレジスタ構成を利用するのであれば、次のように#define定義と併用すると便利です。このときは識別子axも大文字にするとよいでしょう。

```
#define AH h.ah        // この定義をしておくと
#define AL h.al

rg.AH = 0x1A;          // このようにrg.h.xxより簡潔に記述できる
rg.AL = 0x2B;
```

第15章

プリプロセッサ

C Quick Reference

114 前処理指令

プリプロセッサ1

Cのコンパイラは、ソースプログラムを解析する前に、テキストレベルでいろいろな前処理を行ないます。たとえば別のファイルを読み込んだり、文字列を置き換えたりしてくれます。この前処理を行なうのが**プリプロセッサ**(pre：前もって processor：処理するもの)です。プリプロセッサの仕事は、

- 他のプログラムを併合する
- テキスト置き換えを行なう
- コンパイラにコンパイル条件を与える
- コンパイラにその他の情報を伝える

といったものです。前処理の指示は**前処理指令**(プリプロセッサ指令)で行ないます。前処理指令には次のものがあります。

前処理指令

指令	説明
#include	指定されたファイルを挿入する
#define	マクロ定義を行なう
#undef	マクロ定義を取り消す
#if #ifdef #ifndef	条件つきコンパイルを行なう
#elif #else #endif	同上
defined	#if #elifの中で判定処理に用いる単項演算子
#line	行番号のつけ替えを行なう
#pragma	コンパイラへのオプション指示(処理系依存)
#error	前処理時のエラー表示を行なう
#	#defineの中で使用時：仮引数の文字列化 単独使用時：空指令であり何も実行しない
##	字句の連結を行なう

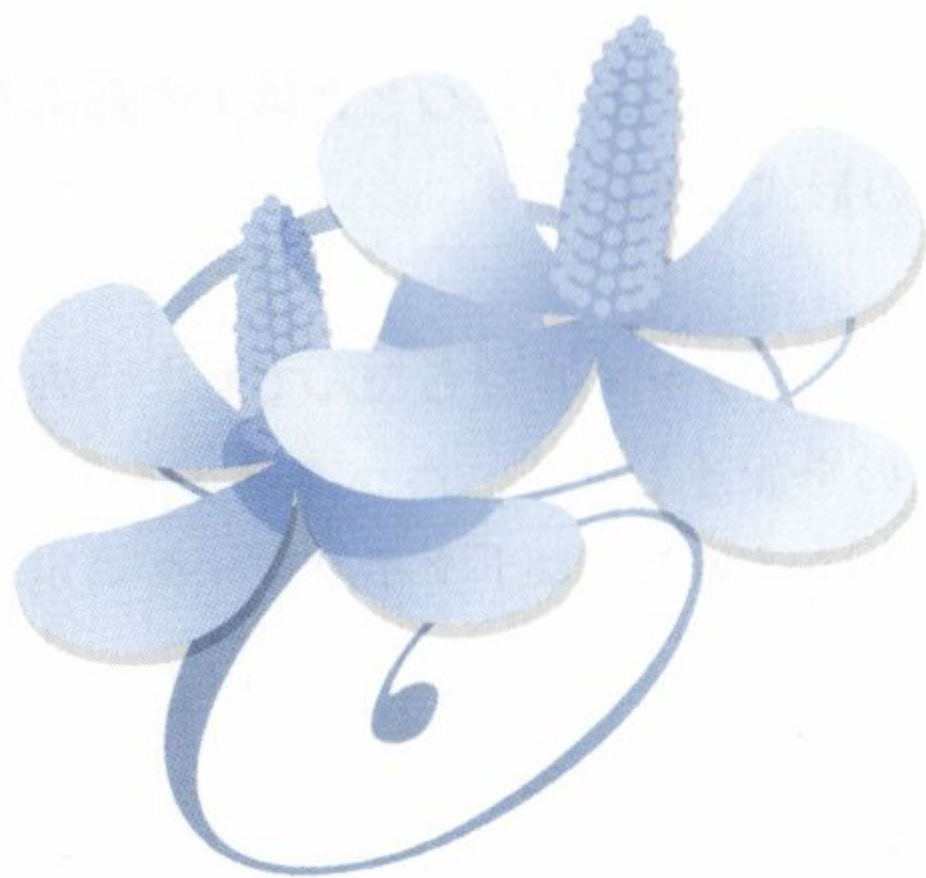
これらの前処理指令は次のように記述します。縦線は行左端を意味します。

```
#define AAA 111            ——(1)標準の記述スタイル
   #define BBB 222         ——(2)左に空白があってもよい
   #   define CCC 333      ——(3)#の前後に空白があってもよい
#define DDD ¥              ——(4)末尾に¥を置くと行継続処理になる
444
```

前処理指令は行単位で有効です。Cの通常の文のように';'がくるまで、何行にも渡っても有効ということはありません。前処理指令は「Cの文」ではないので';'は不要です。入れても単なる文字として処理されます。

(1)は標準的な記述スタイルです。

(2)や(3)のように、'#'の前後に空白を入れることもできます。

(4)のように¥で終わる行は次の行と併合されます。この場合、「行末に'¥'があったら、その'¥'自身と、続く改行文字をないものとみなす」という処理が行なわれます。改行後の先行空白も意味をもちます。

115 ファイルの挿入 #include プリプロセッサ2

#includeの書式

```
#include <文字列>
#include "文字列"
#include 前処理字句列
```

注：「文字列」部分ではヘッダまたはソースファイルを指定する。

#include指令は指定されたヘッダファイル（ヘッダ）またはソースファイルを読み込みます。そしてあたかもはじめからそのファイルの内容がそこに記述してあったかのように処理します。

ファイルの指定にはふたつの方法があります。

```
#include <xxx.h>       ――(1)標準のディレクトリにあるxxx.hを読み込む
#include "xxx.h"       ――(2)処理系定義の探索手順の場所にあるxxx.hを読み込む
                          そこにないときは「#include <xxx.h>」として探す
```

(1)の指定を行なうと、あらかじめ指定されている標準ディレクトリからファイルを探します。「標準ディレクトリ」をどのように指定しておくか（パス設定という）は、C処理系によって決められています。(1)の形式は、C処理系が用意しているファイルを読み込むのに使用します。

一方、(2)の形式は、まず処理系定義の探索手順にもとづいてファイルを探し、そこにないときは記述を<ファイル名>とみなして、もう一度ファイルを探します。このときは標準ディレクトリを探すことになります。

「処理系定義の探索手順」は、通常は「まずカレントディレクトリを探す」ようになっているのが普通です。

一般に、処理系提供のファイルを読み込むときは(1)の形式を、自作のファイルを読み込み指示するときは(2)の形式を使います。

読み込むファイルはテキストファイルならなんでもよいのですが、一般にヘッダファイルと呼ばれているものを記述します。ヘッダファイルには、.hという拡張子をつけるのが慣例で、その中には関数プロトタイプやマクロ定義などがふくまれるのが普通です。

マクロ文字列対応の#include

#include指令は、他のマクロ置換を期待した記述をすることができます。これは「#include 前処理字句列」というスタイルをしています。この前処理字句列は、一般のマクロ処理で、適切なファイル名に置き換えられるように記述します。次に例を示します。

```
#define PRGTYP 1

#if PRGTYP==1
    #define FILENAME "mydos.h"
#elif PRGTYP==2
    #define FILENAME "mywin.h"
#else
    #define FILENAME "myunx.h"
#endif

#include FILENAME
```

PRGTYPの値によってFILENAMEを設定する

FILENAMEで示すファイルをインクルードする

#includeのネスト

#include指令はネストすることができます。つまり#includeでインクルードしたファイルの中に、また#includeが書いてあってもかまいません。ネストの深さ（レベル）は最低でも8（C99では15）であることが規格要請です。

note　ヘッダとヘッダファイル

C言語規格によれば**#include <xxx>**となっているとき、**<>**で囲まれた表現が、必ずしもファイル名である必要はない。これは、たとえば**#include <xxx>**と書くことによって、内部的に対応機能を用意する（内部機能をオンにする）といった役目で用いることができることを意味している。そのため厳密には「ヘッダファイル」ではなく「ヘッダ」が正しい呼称となる。本書は慣例的にヘッダファイルという用語を使っている。

116 マクロ定義 #define

プリプロセッサ3

#defineの書式

```
#define 識別子 置換要素並び                    ——オブジェクト形式マクロ
#define 識別子(仮引数識別子並び) 置換要素並び    ——関数形式マクロ
```

#defineは識別子の置き換えを行ないます。置き換えには単純なテキスト置換を行なうオブジェクト形式マクロと、引数をふくむ置換を行なう関数形式マクロがあります。このように記述したものをマクロ定義といいます。defineに続けて置かれる識別子を「マクロ名」といいます。

オブジェクト形式マクロ

オブジェクト形式マクロは単純な文字列置換に用いられます。「オブジェクト形式」とは、

——オブジェクト表現のようなスタイルで記述する

ということです。次に記述例を示します。

```
#define SIZE1    100              // (1)
#define SIZE2    200              // (2)
#define MEMSIZE  SIZE1+SIZE2      // (3)
#define MEMSIZEB (SIZE1+SIZE2)    // (4)
#define MY_STR   "abcde"          // (5)
#define MY_CH    'A'              // (6)
#define begin    {                // (7)
#define end      }                // (8)
#define then                      // (9)

n = SIZE1;              // (a) 置換結果:n = 100;
n = MEMSIZE;            // (b) 置換結果:n = 100+200;
n = MEMSIZE * 2;        // (c) 置換結果:n = 100+200 * 2;
n = MEMSIZEB * 2;       // (d) 置換結果:n = (100+200) * 2;
puts(MY_STR);           // (e) 置換結果:puts("abcde");
putchar(MY_CH);         // (f) 置換結果:putchar('A');
if (flg) begin a = 10; b = 20; end
                        // (g) 置換結果:if (flg) { a = 10; b = 20; }
if (flg) then c = 30;
                        // (h) 置換結果:if (flg)  c = 30;
```

(1)と(2)は標準的な記述です。(3)と(4)は他のマクロ名を利用しています。マクロ名はC規則の識別子を記述しますが、置換要素並びには記号文字があってもかまいません。(3)〜(8)はその例です。

(a)〜(g)は利用例です。このように置換されることを「マクロ展開された」と表現します。置換要素並びはなくてもかまいません。その場合はマクロ名が単に削除されます。(9)と(h)がその例です。

マクロ展開は、単にテキストの単純置き換えをするだけなので、必要に応じて防御的記述をすることが必要です。(3)の定義をして(c)の利用をすると、たぶん異なる意図になります。防御的に(4)の記述をすると、(d)のように、意図どおりに展開されます。

再走査と再置換

マクロ置換処理を行なったあと、再びテキストを調べ、さらにマクロ置換可能な要素があれば再置換されます。このため、マクロ名位置の前後関係に依存しない記述ができます。これは関数形式マクロの場合も同じです。

```
#define SIZE1 SIZE2*2      // (1) 100*2に置換される
#define SIZE2 100          // (2)
#define SIZE3 SIZE3+200    // (3)
```

この例で(1)の記述は、後続記述の(2)も反映します。また無限に繰り返されることになる置換は、実行されません。(3)がそれに相当します。

関数形式マクロ

関数形式マクロはあたかも関数のように、引数つきでマクロ定義をします。関数と異なるのは、その利用位置にマクロ定義された文字列を埋め込むことです。

基本的な記述例を示します。

```
#define mul(a,b)    a*b
#define maxdt(a,b) (a>b) ? a : b
#define putd(n)     printf("%d¥n", n)

a = mul(10, 20);      // 置換結果：a = 10*20;
a = maxdt(n1, n2);    // 置換結果：a = (n1>n2) ? n1 : n2;
putd(a);              // 置換結果：printf("%d¥n", a);
```

117 マクロ定義用演算子

プリプロセッサ4

#演算子

#演算子は引数を「**文字列リテラル化**」する演算子です。引数名を文字列として反映したいときに便利です。次に利用例を示します。

```
#define DBGOUT(n) printf(#n "の値は%dです¥n", n)

dt = 100;
DBGOUT(dt);      // (1) 置換結果：printf("dt" "の値は%dです¥n", dt);
                 // 出力：dtの値は100です
DBGOUT(dt+20);
                 // (2) 置換結果：printf("dt+20" "の値は%dです¥n", dt+20);
                 // 出力：dt+20の値は120です
```

この例で#nが引数を文字列リテラル化します。仮引数がdtのとき、生成される文字列は「"dt" "の値は%dです¥n"」のようになります。Cコンパイラは「隣接する文字列リテラルは結合する」という処理をするので、最終的に「"dtの値は%dです¥n"」と解釈されます。

また、引数まわりに空白があるときは、

——両端の空白類は削除され、間の空白類は1個の空白にする

というルールで処理されます。次のようになります。

記述

```
DBGOUT(   dt   +   20   );
```

置換結果

```
printf("dt + 20" "の値は%dです¥n", dt + 20);
```

##演算子

##演算子は置換文字列の中に埋もれた引数文字列を、前後の字句と一体化(字句の連結)する演算子です。次に例を示します。

```
#define def1(dt) aaa##dt##bbb       // (1)
#define def2(dt) aaa ## dt ## bbb   // (2)

int def1(ABC);                      // 置換結果：int aaaABCbbb;
int def2(DEF);                      // 置換結果：int aaaDEFbbb;
```

(1)の記述で、置換要素は##で区切られ、「aaa」「dt」「bbb」に分けられます。そして字句dtは引数なので、引数テキストと置換されます。(2)のように##の前後に空白を入れてもかまいません。この##は区切りの役目をもちます。区切りなので置換要素の両端に置くことはできません。

```
#define def3(dt) ##dt##aaaa    ――エラー。dt##aaaaと書けばよい
#define def4(dt) aaaa##dt##    ――エラー。aaaa##dtと書けばよい
```

次の例はラベルを番号だけで指定できるようにしています。

```
#define label(n) LABEL##n

void test(void)
{
    puts("aaa");
    goto label(5);  // 置換結果：goto LABEL5;
    puts("bbb");
label(5):           // 置換結果：LABEL5:
    puts("ccc");
}
```

マクロ記述の注意点

マクロ置換は、C文法とは関係なく、字面上の処理をするだけなので、プログラマの意図のとおりに展開されないことがあります。注意ポイントを示します。

■①かっこをつける

不正例

```
定義：#define mul(a,b) a*b
使用：a = mul(x+100, y+200);
展開：a = x+100*y+200;   // 100*yを先に計算してしまう
```

対策例

```
定義：#define mul(a,b) (a)*(b)
使用：a = mul(x+100, y+200);
展開：a = (x+100)*(y+200);
```

②かっこを多重につける

不正例

```
定義：#define add(a,b) (a)+(b)
使用：a = add(x, y) * 300;
展開：a = (x)+(y) * 300;     // (y)*300を先に計算してしまう
```

対策例

```
定義：#define add(a,b) ((a)+(b))
使用：a = add(x, y) * 300;
展開：a = ((x)+(y)) * 300;
```

③かっこをつけても防ぎきれない例

不正例

```
定義：#define sqr(a) ((a)*(a))
使用：a = sqr(++x);          // ++xを1回評価する意図だが
展開：a = ((++x)*(++x));     // 2回評価されてしまう
```

この③の例はプログラマが「こんなマクロの使い方をしないように注意する」という対応しかありません。このようにマクロ定義は信頼性が関数より劣りますから、意図しない結果が起きないように厳重に配慮をしておくことが大切です。あるいは単純利用のみを行なうのが安全です。

なおC99では、インライン関数機能が追加になっています。インライン関数は、表面的にはマクロのように利用できますが、機能は関数であり、通常の関数ルールで引数が処理されるので、上記の「++xの2回評価」のような問題はおきません。→「095 インライン関数」(p. 135)

118 定義済みのマクロ名

プリプロセッサ5

Cではプログラマの便宜のために、処理系側で最初から用意されている**定義済みのマクロ名**があります。これを使うと、コンパイルした日付や時刻をプログラムの中で使用することができます。定義済みマクロ名には次のものがあります。

マクロ名	説明	
`__FILE__`	現在処理中のソースファイルの名前を示す文字列	
`__TIME__`	コンパイル時刻を示す文字列	
`__DATE__`	コンパイル日付を示す文字列	
`__LINE__`	ソースファイル中の現位置の行番号を示す10進数。先頭行は1	
`__STDC__`	処理系がC規格合致であるとき定数1	
`__STDC_HOSTED__`	規格合致ホスト処理系であるとき定数1。[C89(追補1)]で追加	
`__STDC_VERSION__`	C89(追補1)に合致なら定数199409L。C99に合致なら定数199901L	C99
`__STDC_IEC_559__`	附属書F(IEC 60559浮動小数点演算)の規定に合致なら定数1	C99
`__STDC_IEC_559_COMPLEX__`	附属書G(IEC 60559互換複素数演算)の規定に合致なら定数1	C99
`__STDC_ISO_10646__`	`wchar_t`型の値が、ISO/IEC 10646に準拠していれば`yyyymmL`形式の定数	C99

利用例

```
printf("[%s] [%s] [%s] [%d]¥n", __FILE__, __TIME__, __DATE__,
__LINE__);
                    // 出力例：[test1.c] [10:26:48] [May  2 2010] [25]
```

119 再定義とマクロ取り消し #undef プリプロセッサ6

#undefの書式

```
#undef マクロ名
```

#undefは、すでに定義されているマクロ名を未定義状態に戻すときに使います。
マクロ定義は内容が同一であれば再定義しても警告されません。ここで同一とは、

- 置換要素並びに使われている文字が同一である
- 空白の有無も同一である（ただし連続空白と単一空白は同一とみなされる）
- 関数形式マクロの場合は仮引数の文字と個数も同一である

ということです。次に記述例を示します。

```
#define def1 (10+20)        // 初回定義
#define def1 (10+20)        // 再定義 OK まったく同一
#define def1 ( 10+20)       // 再定義 NG 空白が入っている
#define def1 (55+20)        // 再定義 NG 文字が違う

#define ps(s) puts(s)       // 初回定義
#define ps(s) puts(s)       // 再定義 OK まったく同一
#define ps(z) puts(z)       // 再定義 NG 文字が違う
#define ps(s) puts( s )     // 再定義 NG 空白が入っている
```

あらかじめ#undefで定義取り消しを行なえば、新しい内容のマクロ定義を行なっても無警告になります。次のように使います。

```
#define SIZE 100            // 最初の定義
n = SIZE;                   // 置換結果：n = 100;
#undef SIZE                 // 一度定義を取り消して
#define SIZE 200            // 再定義する
n = SIZE;                   // 置換結果：n = 200;
```

C規格では、未定義のマクロ名に対して#undefを使用してもエラーにならないことが保証されています。ですから定義しているかどうかあいまいなマクロ名に対して「念のために#undefしておく」といった処理も可能です。

120 関数形式マクロの可変個引数 プリプロセッサ7

C99では、関数形式マクロで、**可変個の引数**を記述できます。

```
...             ——仮引数の可変個を表現
__VA_ARGS__     ——置換要素側で可変個引数を表現
```

このふたつの表現はprintf()のような、もともと可変個引数を受け付ける関数とともに利用すると効果を発揮します。

次の例はFor～Endで処理できる制御文を定義しています。for文の中でカンマ演算子を使うと、引数個数が違ってくるので、可変個引数処理を利用しています。

```
#define For(...)  for (__VA_ARGS__) {
#define End       }

For (sum=0,n=1; n<=10; n++) // 置換結果：for (sum=0,n=1; n<=10; n++) {
    sum += n;
End                         // 置換結果：}
```

先頭に固定引数をもち、末尾側に...を記述することもできます。その場合...は「残りの引数全部」を意味します。次の例は少し癖がありますが、生成文字列長を指定できるsprintf()関数です。C99規格で追加になったsnprintf()関数に似ています。

```
#define n_sprintf(s,n, ...) sprintf(buf,__VA_ARGS__),*s='¥0',
➡strncat(s,buf,n)
char buf[1000];
char ss[80];
...
n_sprintf(ss, 10, "value=%d", 123456);   // 生成文字列の長さは10まで
puts(ss);                                // 出力：value=1234
```

参考：snprintf関数の書式

```
int snprintf(char * restrict s, size_t n, const char * restrict
format, ...);
```

121 条件つきコンパイル #if

プリプロセッサ8

#if #else #elif #endifの書式

```
#if 定数式1
      この部分をコンパイル
#elif 定数式2
      この部分をコンパイル
...
#elif 定数式n
      この部分をコンパイル
#else
      この部分をコンパイル
#endif
```

#if #else #elif #endifの各前処理指令は**条件つきコンパイル**に使用します。条件つきコンパイルでは、定数式を評価し、はじめに真となった#ifまたは#elif部分のテキストをコンパイルします。

どの定数式も真でないときは、#elseの部分をコンパイルします。

#elifは0個以上記述できます。#elseは必要なければ省略できます。

次に記述例を示します。

```
#define MEMSIZE 640          ——この値を書き換えると

#if    MEMSIZE == 640
       char buf[60000];
#elif MEMSIZE >= 480
       char buf[44000];      } 確保されるbufの配列長が変わる
#else
       char buf[32000];
#endif
```

次の記述はデバッグ用によく使われます。

```
#define MYDEBUG 1                  // デバッグ表示不要のときは0にする
...
#if MYDEBUG                        // MYDEBUGが真なら
    printf("debug: a=%d¥n", a);    // 表示する
#endif
```

定数式は通常の比較・等価演算子が使えますから、

```
#if MODE==2 || MODE==3 ——複雑な評価をする
```

のような記述も可能です。

note #ifを強力コメント機能として使う

#ifは条件つきコンパイルを行なうものだが、強力なコメント機能としても有用である。**/*〜*/**があっても、指定範囲全体を強引にコメント化できる。次に例を示す。

```
#if 0
    for (i=0; i<=4; i++)
        putchar(ss[i]); /* 表示する */
    putchar('¥n');
#endif
```
（すべてコメントになる）

このとき先頭行を「**#if 1**」にすれば、簡単にコメント解除になる。

またコードの候補が2組あり、動作を切り分け試験したいときには、次のように記述することができる。これも便利である。

```
#if 1               // ここを1/0と変更するだけで切り分けコンパイルする
    複数行のコード  // 1のときコンパイル
#else
    複数行のコード  // 0のときコンパイル
#endif
```

122 定義ありの確認1 #ifdef #ifndef プリプロセッサ9

#ifdef #ifndefの書式

```
#ifdef  識別子    ――識別子が定義されていたら
#ifndef 識別子    ――識別子が未定義なら
```

#ifdefと#ifndefは「ある識別子が定義されていたら」「ある識別子が未定義なら」という判定を行なうものです。指定した識別子が、その時点でマクロ名として登録されていれば真になります。定義されているかどうかだけを判定対象にします。

```
#define MYDEBUG                     // マクロ名の定義だけでよい。その値は不要
...
#ifdef MYDEBUG                      // MYDEBUGというマクロ名が定義されていたら
    printf("debug: a=%d¥n", a);     // デバッグ情報を表示する
#endif
```

次の記述は、あるヘッダファイルが何度も読み込まれないようにガードをするために使われる手法です。「インクルードガード」と呼ばれます。

このようにガードを行なうと、「いろいろな記述」の部分が、一度しか読み込まれません。

```
#ifndef MYHEAD_H      // まだ未定義なら
#define MYHEAD_H      // 定義済みにする
(いろいろな記述)
#endif
```

123 定義ありの確認2 defined演算子 プリプロセッサ10

defined演算子の書式

```
#if defined 識別子        ——識別子が定義されていたら
#if defined(識別子)       ——同上。かっこをつけてもよい

#if !defined 識別子       ——識別子が定義されていなかったら
#if !defined(識別子)      ——同上。かっこをつけてもよい
```

#ifコマンドとdefined単項演算子を組み合わせる方法でも、「ある識別子が定義されていたら」「ある識別子が未定義なら」という記述をすることができます。この**defined演算子**は、#if #elif #else #endifと組み合わせて用います。識別子にはかっこをつけてもかまいません。

先の#ifdefの記述例をdefined形式にすると次のようになります。

```
#ifdef MYDEBUG              // #ifdefで記述
#if defined MYDEBUG         // 同じ機能になる
```

definedとともに()を用いるかどうかは自由です。またdefinedを使うと、複雑な定義確認もできます。

```
#if defined AAA || defined BBB || MODE==2        // どれかが真なら
#if defined(AAA) || defined(BBB) || MODE==2      // 同上
```

124 行番号とファイル名の変更 #line プリプロセッサ11

#lineの書式

```
#line 行番号                  ——__LINE__の内容を変更する
#line 行番号 "ファイル名"     ——__LINE__と__FILE__の内容を変更する
#line 前処理字句列            ——前処理字句列で処理する
```

コンパイル実行中の行番号とソースファイル名は定義済みマクロ名である__LINE__と__FILE__が記憶しています。**#line**は、コンパイル中の記憶されている行番号とファイル名を指定したものに変えます。ファイル名を変更する必要がないときは、行番号だけを指定します。指定した行番号は次の行から有効になります。

```
#line 100 "myprg.c"   ——こう書くとファイル名がmyprg.cになり
int a;                ——ここが100行目になる
...
#line 200             ——こう書くとファイル名は同じで
a = 5;                ——ここが200行目になる
...
```

このあと行番号は(__LINE__の内容は)ひとつずつ増えて行きます。#line指令で行番号とファイル名を変更すると、以降で発生したエラーメッセージの中の行番号とファイル名に反映されます。

この機能は、「ツールを使ってaa.cをbb.cに加工し、bb.cをコンパイルする」というケースで、表示されるエラーメッセージに元のファイル名と行位置を反映させるときに重宝します。

実例として、Cソースプログラムの前処理だけを行なうと(Visual C++では/P指定をする)、生成された前処理済みファイルの中には、多くの#line指令が挿入されます。

また「マクロ文字列対応の#include」(p. 165)で説明したのと同じように、「#line 前処理字句列」というスタイルも用いることができます。たとえば次のようにします。

```
#define LINEDATA 1000 "myprg.c"   ——あらかじめこの設定をすると
#line LINEDATA                    ——#line 1000 "myprg.c"とみなされる
```

125 プラグマ指令 #pragma

プリプロセッサ12

#pragmaの書式

```
#pragma 前処理字句列
```

#pragma(pragmatize=現実化する)はそのコンパイラにあらかじめ用意されている特殊な機能(処理系独自の機能)を実行するように、コンパイラに指示を与える指令です。通常、コンパイラへの指示はコマンドラインからスイッチを使って行なわれますが、#pragmaでは、

——ソースコード側からコンパイラに指示を与える

ということができます。どのような指示を与えられるかは、その言語開発者がどのような指示子を用意したかによります。その「指示子」の内容についてはC規格は関知しません(後述の標準プラグマを除く)。処理系が認識できない指示子が使われたときは、そのプラグマ記述は、

——不明なプラグマがありました

のような警告は受けるかもしれませんが、エラーではなく単に無視されます。

ある処理系は次のプラグマ指令を受け付けます。

```
#pragma optimize ～   ——最適化方法の指定
#pragma warning ～    ——警告方法の指定
```

C99では定型の記述をする標準のプラグマ指定として、次のものが用意されました。(1)(2)は浮動小数点演算の処理方法を、(3)は複素数演算の処理方法を指定するものです。

```
#pragma STDC FP_CONTRACT 状態切替指定        ——(1)
#pragma STDC FENV_ACCESS 状態切替指定        ——(2)
#pragma STDC CX_LIMITED_RANGE 状態切替指定   ——(3)
```

注：状態切替指定は、ON, OFF, DEFAULTのいずれか。

126 プラグマ演算子 _Pragma プリプロセッサ13

C99ではプラグマ指令を特殊な局面でも書きやすくするために、**プラグマ演算子_Pragma**が追加になっています。

```
_Pragma("前処理字句列")   ――このように書くと
#pragma 前処理字句列      ――このように解釈される
```

127 エラー指令 #error プリプロセッサ14

#errorの書式

```
#error 前処理字句列
```

#errorはコンパイル時に独自の診断メッセージを出すのに使用します。通常は#if文とともに用います。次に例を示します。

```
#define MEM_SIZE 35000

#if MEM_SIZE > 30000
#error MEM_SIZEは30000以下で指定してください。
#endif
```

この記述のあるプログラムをコンパイルすると、たとえば、

```
fatal error C1189: #error :  MEM_SIZEは30000以下で指定してください。
```

のような警告メッセージが表示されます。

128 空指令 # プリプロセッサ15

プログラムの中に制御命令をもたない'#'だけの行があると、その行は単に無視されます。空行と同じです。

第16章

標準入出力関数

C Quick Reference

129 標準入力と標準出力

標準入出力関数 1

Cでは、キーボード、ディスプレイ、ファイルなどとのデータ入出力は、標準ライブラリ関数を用いて行ないます。ここでは入出力関数の利用方法を、用途別に簡潔に整理して示します。入出力関数を用いるときは、「#include <stdio.h>」を記述しておきます。

一般的な用語として「コンソール入出力」という表現が用いられます。これは「キーボードから何か入力し、ディスプレイに何か出力する」という処理を意味します。しかし厳密には「コンソール入出力」という表現は正しくありません。正しくは、

――**「標準入力」**から何か入力し、**「標準出力」**に何か出力する

となります。そして特に「指定」がないとき、

標準入力 → **キーボード**
標準出力 → **ディスプレイ**

と認識されます。ではその「指定」とは何かというとリダイレクト指示です。たとえば、「標準入力からデータを入力し何かの処理をして標準出力に出力する」という仕事を行なう次のプログラムがあったとします。

```
/* myprog.c */
#include <stdio.h>
int main(void)
{
    char s[100];
    gets(s);        // 1行入力
    puts(s);        // 1行出力
    return 0;
}
```

このプログラムは次のように実行させることができます。

```
myprg                  ――入力：キーボード      出力：画面
myprg < a.txt          ――入力：a.txtファイル   出力：画面
myprg > b.txt          ――入力：キーボード      出力：b.txtファイル
myprg < a.txt > b.txt  ――入力：a.txtファイル   出力：b.txtファイル
```

この標準の入出力先は「標準ストリーム」と呼ばれ、次の名前で利用できるようになっています。これについては「143 標準ストリーム」(p. 208)で詳しく説明します。

標準入力 → **stdin**
標準出力 → **stdout**

130 文字の入出力

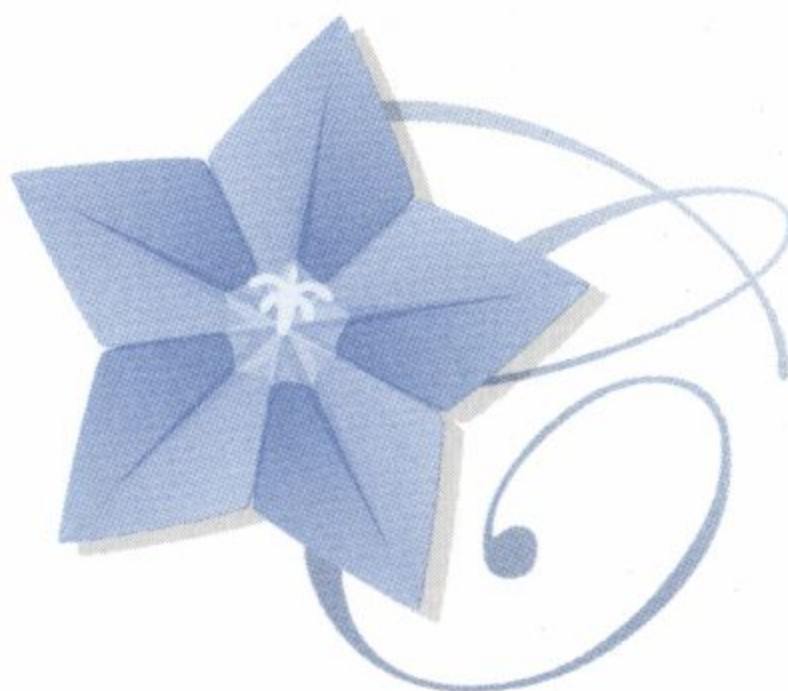
標準入出力関数2

1文字入力はgetchar()関数で、1文字出力はputchar()関数で行ないます。入力終了時(WindowsはCTRL+Z, UNIXはCTRL+D)に、getchar()関数は記号定数EOFを返します。

例：キーボードから1文字入力して、それを出力する

```
int ch;
ch = getchar();  // 1文字入力
putchar(ch);     // それを出力
putchar('A');    // 'A'を出力
```

例：キーボードから入力終了になるまで1文字入力して、それを出力する

```
int ch;
while ((ch=getchar()) != EOF)
    putchar(ch);
```

131 1行文字列の入出力

標準入出力関数3

文字列の1行入力はgets()関数で、1行出力はputs()関数で行ないます。入力終了時に、gets()関数は記号定数NULLを返します。

gets()関数で指定するchar型配列は、入力文字列を確実に格納できる長さにしておく必要があります。puts()関数は出力後に改行を行ないます。

例：キーボードから文字列入力して、それを出力する

```
char s[100];
gets(s);          // 1行入力
puts(s);          // それを出力して改行する
puts("abcd");     // "abcd"と出力して改行する
```

例：キーボードから入力終了になるまで文字列入力して、それを出力する

```
char s[100];
while (gets(s) != NULL)
    puts(s);
```

note getsよりfgets

gets()関数は想定よりも長い文字列が入力されると危険であることは、広く認識されている。そのため重要なプログラムの場合は、読み込み最大文字数を指定できる**fgets()**関数と、標準入力を示す**stdin**を使って、次のように記述する方が安全である。

```
char s[80];
fgets(s, 80, stdin);
            ――stdinから（¥0をふくめて）最大80文字を読み込んでsに格納
```

この例でもし79文字を超える入力があった場合、その超過分は読み残しになり、次回の読み込みで処理される。

fgets()関数では改行文字（**¥n**）も格納されるので注意。ただし格納可能文字数を超える（上の例では79文字以上）入力があるときは、改行文字は格納されない。

132 書式つき出力1

標準入出力関数4

printfの用法

printf()関数は指定される書式の制御にしたがって、対応する実引数を出力します。書式指定文字列は、出力の整形方法を定めます。printf()関数の引数は次のように記述します。「引数並び」は0個以上を並べます。

printfの書式

```
printf(書式指定文字列, 引数並び)
```

書式指定文字列は二重引用符で囲まれた文字列か、文字配列変数です。基本的な記述例を示します。

```
int    idt = 123;
double ddt = 456.789;
char   *str = "abcde";

printf("Hello");               // (1) Hello と表示
printf("Hello¥n");             // (2) Hello と表示して改行
printf("idt=%d¥n", idt);       // (3) 出力：idt=123
printf("ddt=%f str=%s¥n", ddt, str);
                               // (4) 出力：ddt=456.789000 str=abcde
```

この例で(1)は文字列をそのまま表示します。改行をするときは(2)のように、自分で¥nを入れます。(3)で%dは、対応する実引数idtの値を10進数出力する指定です。(4)の%fは浮動小数点形式で、%sは文字列として出力させるための指定です。

pritnf()関数は「正常なときは出力した文字数、エラーのときは負の値」という戻り値を返します。しかしprintf()の戻り値を利用することはあまりありません。

書式指定文字列の中に「%ではじまる〈変換指定〉」があれば、対応する引数の値を形式変換して出力します。〈変換指定〉以外の文字列はそのまま出力されます。"%"のあとの何文字が変換用の特殊文字となるかは、それぞれの機能によります。

133 書式つき出力2

標準入出力関数5

printfの変換指定

printf()関数で用いる出力用の〈変換指定〉は次のように記述します。

変換指定の書式

% フラグ フィールド幅 .精度 長さ修飾子 変換指定子

ここで変換指示開始の%と変換指定子以外はオプションなので、必要なければ省略できます。最少の変換指定は%dの形式をしています。〈変換指定〉の各要素を次に示します。

■フラグ(省略可能)

-	左詰めで出力(指定がないと右詰め)
+	数値の前に+/-の符号を出力(指定がないと-のみ出力)
空白	正数の場合に先頭に空白を出力して負の場合と位置を揃える
0	フィールド幅の左側への詰め込みに空白ではなく0を使う
#	代替形式で出力。数値の型ごとに出力形式を決める 対応する変換指定子により以下のように効果が変わる **o** :8進データの先頭に**0**をつける **x, X** :16進データの先頭に**0x**または**0X**をつける **a, A, e, E, f, F** :出力に必ず小数点をつける **g, G** :同上で、さらに小数点に続く**0**を省略しない

注:a, A, Fは C99 で追加。

■フィールド幅(省略可能)

数値	出力の幅を指定する データの幅が指定幅より小さいとき、左に空白が詰められる。 データの幅が指定幅より大きいときは、データの桁で出力される
*	対応する引数の値を幅指定とする

■精度(省略可能)

.数値	小数点以下の桁数、または最大表示文字数を指定する 対応変換指定子により以下のとおり効果が変わる(指定値を**n**とする) **d, i, o, u, x, X** :少なくとも**n**個の数字を出力 :数字が足りないときは先行0埋めをする **a, A, e, E, f, F** :小数点の後に**n**個の数字を出力 **g, G** :有効桁数を最大**n**にする **s** :最大表示文字数を**n**にする
.*	対応する引数の値を数値とする

注:ピリオドだけで.と指定された場合は.0とみなす。

注:a, A, Fは C99 で追加。

■長さ修飾子(省略可能)

変換指定子の前につけ、対応引数の型が右側のとおりであることを示す。

h	d, i, o, u, x, X	:shortまたはunsigned short
	n	:short *
hh	d, i, o, u, x, X	:signed charまたはunsigned char
	n	:signed char *
l	d, i, o, u, x, X	:longまたはunsigned long
	n	:long *
	c	:wint_t
	s	:wchar_t * C99
ll	d, i, o, u, x, X	:long longまたはunsigned long long
	n	:long long *
L	e, E, f, g, G	:long double
j	d, i, o, u, x, X	:intmax_tまたはuintmax_t
	n	:intmax_t *
z	d, i, o, u, x, X	:size_tまたはそれに対応する符号つき整数型
	n	:size_tに対応する符号つき整数型へのポインタ
t	d, i, o, u, x, X	:ptrdiff_tまたはそれに対応する符号つき整数型
	n	:ptrdiff_t *

注:hh ll j z tはC99で追加。

■変換指定子

o	8進数で出力する
d, i	10進数で出力する
x, X	16進数で出力する。xは小文字, Xは大文字表示
u	符号なし10進数で出力する
c	文字として出力する
s	文字列として出力する
f, F	10進表記の浮動小数点数として出力する(float, double共) C99ではinf, infinity, nanの出力も行なう C99 fは小文字、Fは大文字表示
e, E	指数形式の浮動小数点数として出力するeは小文字、Eは大文字表示
g, G	g指定は通常はf変換を行なうが、指数部が-4より小さいか、精度以上のときe変換を行なう。GはFかEかの変換を行なう。この変換指定子の意図はf/F変換表示が長くなるとき、自動的にe/E変換を行なうということである
a, A	浮動小数点定数を16進数の[-]0xh.hhhhp±d形式で出力。Aは大文字表示
p	ポインタとして出力する。表示形式は処理系に依存する
n	%nがくるまでに出力された文字の数を対応するint*型引数に書き込む
%	%自身を出力する。"%%"と書くことで'%'を表示する

注:a A FはC99で追加。

134 書式つき出力3

標準入出力関数6

printfの書式指定例

printf()関数のいろいろな変換指定を確認する記述例を示します。出力はC処理系により異なることがあります。

ここでは次の変数が宣言されているとします。

```
char   ch, ss[80];
short  sh;
int    id, n, n1, n2;
long   lng;
double db;
long double ldb;
```

■例1：変換指定子の確認

記述	出力結果	コメント
id = 4080; db = 1024.75;		
printf("o[%o]¥n", id);	o[7760]	8進数
printf("d[%d]¥n", id);	d[4080]	10進数
printf("i[%i]¥n", id);	i[4080]	10進数
printf("x[%x]¥n", id);	x[ff0]	16進数小文字
printf("X[%X]¥n", id);	X[FF0]	16進数大文字
printf("a[%a]¥n", db);	a[0x1.003000p+10]	16進数浮動小数点
id = -5;		int型の-5の内部表現FFFFFFFBを
printf("u[%u]¥n", id);	u[4294967291]	unsigned int型として出力
ch = 'a'; id = 'b';		
printf("c[%c]¥n", ch);	c[a]	文字出力
printf("c[%c]¥n", id);	c[b]	対応する引数はintでもよい
strcpy(ss, "abcde");		
printf("s[%s]¥n", ss);	s[abcde]	文字列出力
db = 123.456;		
printf("f[%f]¥n", db);	f[123.456000]	小数表現
printf("e[%e]¥n", db);	e[1.234560e+002]	指数表現小文字
printf("E[%E]¥n", db);	E[1.234560E+002]	指数表現大文字
db = 0.000345;		
printf("g[%g]¥n", db);	g[0.000345]	指数部-4のときf表現
db = 0.0000345;		
printf("g[%g]¥n", db);	g[3.45e-005]	指数部-5のときe表現
printf("G[%G]¥n", db);	G[3.45E-005]	その大文字表現
db = 123.456;		
printf("g[%g]¥n", db);	g[123.456]	精度以内のときf表現
db = 12345678.9;		
printf("g[%g]¥n", db);	g[1.23457e+007]	精度以上のときe表現
printf("p[%p]¥n", &id);	p[0012FF3C]	変数idのアドレス
printf("123%%456¥n");	123%456	'%'自身を出力

注1：最小の指数部表示桁数をこの処理系(Visual C++)では1.234560e+002のように3桁で表示しているが、1.234560e+02のように2桁で表示する処理系もある。

■例2：長さ修飾子の確認

コード	出力	説明
id = 1234; sh = 5678;		
printf("d [%d]¥n", id);	d [1234]	int型の通常の出力
printf("hd[%hd]¥n", sh);	hd[5678]	short型の出力
lng = 87654321;		
printf("ld[%ld]¥n", lng);	ld[87654321]	long型の出力
ldb = 2.345678;		
printf("Lf[%Lf]¥n", ldb);	Lf[2.345678]	long doubleの出力

注2：データ幅がint==longのシステムでは%dと%ldは同じ効果になる。同様にdouble==long doubleのシステムでは%fと%Lfは同じ効果になる。この事情はscanf()でも同じである。

■例3：フィールド幅と精度の確認

コード	出力	説明
id = 123; n = 10;		
printf("d [%d]¥n", id);	d [123]	標準出力
printf("8d [%8d]¥n", id);	8d [123]	8文字幅
printf("*d [%*d]¥n", n,id);	*d [123]	n文字幅
printf(".10d [%.10d]¥n", id);	.10d [0000000123]	少なくとも10字
db = 12.34567; n1 = 10; n2 = 3;		
printf("f [%f]¥n", db);	f [12.345670]	標準出力
printf("12f [%12f]¥n", db);	12f [12.345670]	12文字幅
printf("12.2f[%12.2f]¥n", db);	12.2f[12.35]	小数点以下2桁
printf("12.0f[%12.0f]¥n", db);	12.0f[12]	小数点以下なし
printf("12.f [%12.f]¥n", db);	12.f [12]	同上
printf("*.*f [%*.*f]¥n", n1,n2,db);	*.*f [12.346]	n1.n2指定
strcpy(ss, "abcdefghij");		
printf(".5s [%.5s]¥n", ss);	.5s [abcde]	最大5文字表示

■例4：フラグの確認

コード	出力	説明
id = 123;		
printf("d [%d]¥n", id);	d [123]	標準出力
printf("-8d[%-8d]¥n", id);	-8d[123]	左詰め8文字幅
printf("08d[%08d]¥n", id);	08d[00000123]	ゼロ・パディング
printf("+d [%+d]¥n", id);	+d [+123]	+符号も表示
printf("_d [% d]¥n", id);	_d [123]	+のときに空白を置く
printf("_d [% d]¥n", -246);	_d [-246]	-のときと位置合わせできる
printf("#o [%#o]¥n", id);	#o [0173]	0つき8進数
printf("#x [%#x]¥n", id);	#x [0x7b]	0xつき16進数

135 書式つき入力1

標準入出力関数7

scanf()関数は指定される書式の制御にしたがって、入力を行ない、対応する実引数の示すオブジェクトに格納します。書式指定文字列は、入力の処理方法を定めます。scanf()は機能としてはprintf()の逆の働きをします。変換指定子などもprintf()のものと共通部分が多くあります。scanf()関数の引数は次のように記述します。

```
scanf(書式指定文字列, 引数並び)
```

書式指定文字列は二重引用符で囲まれた文字列か、文字配列変数です。基本的な記述例を示します。

```
int     idt;
char    str[100];

scanf("%d", &idt);      // (1) 数値を入力してidtに格納
scanf("%s", str);       // (2) 文字列を入力してstrに格納
```

この例で(1)は入力された数値をidtに格納します。格納先はアドレスで指定します。(2)は入力された文字列をstrに格納します。配列名はアドレスなので、そのままstrを引数に使用できます。

scanf()関数は、正常なときは入力した項目数を戻り値にし、ひとつも入力できないまま入力誤りが発生したときは、EOFを戻り値にします。

書式指定文字列で期待した入力データと、実際に入力された入力データとに矛盾が生じた場合は、その時点でscanf()動作を終了します(入力の失敗)。たとえば、

```
scanf("%d", &idt)    ――変数idtに10進数字読み込む
```

に対して「a」という文字を入力すると、入力の失敗となります。このときキーボードから入力されたデータはストリーム領域に残されたままになります。そのデータは次のscanf(), getchar(), gets()などの入力系関数で読み込むことができます。同じ記述のscanf()で再読み込みすると、再び失敗しますので工夫が必要です。入力失敗が生じたかどうかは戻り値で確認できます。

書式指定文字列の中には次のものを書くことができます。〈変換指定〉で'%'のあとの何文字が意味をもつかは、それぞれの機能によります。

%以外の文字列は入力の区切りとして用いられます。入力の区切り文字列が特に指定されない場合は、空白類文字が入力の区切りになります。特別な指定を除いて(%cなど)、scanf()で空白をふくむ文字列を入力することはできません。

書式指定文字列

空白	書式文字列中に空白を置くと入力データ中の空白文字を読み捨てる これは独自の区切り文字を指定するときに効果がある 連続空白はひとつの空白と効果が同じである
一般の文字(%を除く)	既定の空白文字に代わって複数の入力の区切り文字とする
%ではじまる〈変換指定〉	変換を指示する

scanf()関数の読み取り処理を整理すると次のようになります。

(1)書式内の指令が[c nでないときは、まず空白類文字を読み飛ばす
(2)次に指令にしたがって入力を読み取る
指令と入力が不一致の場合は読み込みを終了する
(3)指令が空白類文字のときは、その時点で再び空白類文字の読み飛ばしを行なう
空白類文字が複数連続していても、単一の空白類文字と効果は同じである
(4)変換指定でも空白類文字でもない文字があるときは、区切り指定であり、その文字を読み飛ばす
区切りが不正のときも読み込みを終了する

136 書式つき入力2

標準入出力関数8

scanfの変換指定

scanf()関数で用いる入力用の〈変換指定〉は次のように記述します。

変換指定の書式

% 代入抑止 フィールド幅 長さ修飾子 変換指定子

ここで変換指示開始(%)と変換指定子以外はオプションなので、必要なければ省略できます。

〈変換指定〉の各要素を次に示します。

■代入抑止(省略可能)

*	代入を抑止する。対応する位置にあるデータを読み捨てる

■フィールド幅(省略可能)

数値	入力データをこの幅で区切る。入力データの幅が指定より小さいときは通常の区切り文字で区切られる

■長さ修飾子(省略可能)

変換指定子の前につけ、対応引数の型が右側のとおりであることを示す。

h	d, i, o, u, x, X, n	:short *またはunsigned short *
l	d, i, o, u, x, X, n	:long *またはunsigned long *
	a, A, e, E, f, F, g, G	:double *
	c, s, [	:wchar_t *
L	a, A, e, E, f, F, g, G	:long double *
hh	d, i, o, u, x, X, n	:signed char *またはunsigned char *
ll	d, i, o, u, x, X, n	:long long *またはunsigned long long *
j	d, i, o, u, x, X, n	:intmax_t *またはuintmax_t *
z	d, i, o, u, x, X, n	:size_t *または対応する符号つき整数型へのポインタ
t	d, i, o, u, x, X, n	:ptrdiff_t *または対応する符号なし整数型へのポインタ

注:hh ll j z tはC99で追加。

■変換指定子

o	符号省略可能な8進整数変換 対応引数は符号なし整数型へのポインタ
d	符号省略可能な10進整数変換 対応引数は符号つき整数型へのポインタ
i	符号省略可能な整数変換 **0123**(8進数)、**123**(10進数)、**0xabc**(16進数)のスタイルを認識する 対応引数は符号つき整数型へのポインタ
x	符号省略可能な16進整数変換 入力の先頭に**0x**か**0X**がついてもつかなくてもよい 対応引数は符号なし整数型へのポインタ
u	符号省略可能な10進整数変換 対応引数は符号なし整数型へのポインタ
c	1個の文字を読み込む フィールド幅指定があれば、指定個数の文字を読み込む このとき配列の最後に'**¥0**'は付加されない 空白文字も読み込むことができる(空白スキップはしない)
s	文字列を入力する。空白に出合うと入力終了になる 空白をふくむ文字列は入力できない 入力先配列の最後に'**¥0**'を付加する
a, e, f, g	符号省略可能な浮動小数点数変換。これらの指定子は**scanf()**では同一機能 対応引数は浮動小数点数型へのポインタ C99では**0x, 0X**ではじまる16進実数値も認められる
p	ポインタ値への変換 読み込まれる文字並びは**printf()**の**%p**変換で出力される形式と同じ 対応引数は**void****型である
n	**%n**がくるまでに入力された文字数を対応する**int***型引数に書き込む 以下のプログラムで"**abcde**"を入力すると**a**の値は5になる **char ss[80]; int a;** **scanf("%s%n", ss, &a);**
%	入力された'%'を読み捨てる。実際の指定は%%で行なう
[...]	スキャンセット(**scanset**)と呼ばれる文字集合を**[]**で囲んだものである スキャンセットに合致する文字だけで構成される文字列を読み込む スキャンセットに合致しない文字に出合うと入力を終了する 例:**scanf("%[abcd]", ss);** → **abcd**だけで構成される文字列を読み込む。オプションとして反転文字列指定(**^**)がある 例:**scanf("%[^abcd]", ss);** → **abcd**以外の文字で構成される文字列を読み込む 補足:C規格の標準ではないが範囲文字列指定(**-**)の使える処理系が多い 例:**scanf("%[a-d]", ss);** → **abcd**だけで構成される文字列を読み込む 例:**scanf("%[0-9a-fA-FxX]", ss);** → **0123456789abcdefABCDEFxX**だけで構成される文字列を読み込む 例:**scanf("%[-a-d]", ss);** → **-**と**abcd**だけで構成される文字列を読み込む

注1:変換指定子A, E, F, G, Xも有効であり、それぞれa, e, f, g, xと同じ動作になる。

注2:a A FはC99で追加。

137 書式つき入力3

標準入出力関数9

入力データの区切り

ひとつのscanf()で複数のデータを入力できます。そのとき**入力データの区切り**は空白、タブ、改行の空白類文字で行なわれます。また特別の文字で区切りたいときは書式文字列中にそれを書きます。たとえば「カンマ」を置けば、それが区切り記号になります。

```
scanf("%d%d%d", &a, &b, &c);      // (1) 入力例：10 20 30
scanf("%d,%d,%d", &a, &b, &c);    // (2) 入力例：10,20,30
```

ここで(2)のときに「10 , 20 , 30」のような空白入り入力にすると正しく読み込みません。(2)の変換指示は、

――%dの直後に区切り文字,があること

を期待するからです(%dの実行直前にはデフォルトで空白スキップを行なう)。区切り文字の前後に空白を許可する(空白があってもなくてもよい)場合は、

```
scanf("%d , %d , %d", &a, &b, &c);    // 区切り文字の前後の空白を許可する指定
```

にするとよいでしょう。

空白類文字の中にはタブや改行もふくまれます。ですから"%d%d%d"の入力指定のときに、

入力例1

```
10 20 30↵
```

入力例2

```
10↵
20↵
30↵
```

のどちらのスタイルでも正しく入力できることを示しています。

また、(1)の記述は、

```
scanf("%d", &a);
scanf("%d", &b);
scanf("%d", &c);
```

と書くことと動作は同じです。スタイルの点では単純な記述の方が明快です。

float型とdouble型の変換指定

float型とdouble型の変換指定子は、printf()のときはともに"%f"を使用できます。scanf()のときは"%f"と"%lf"に使い分けするので注意が必要です。

```
float  fd; double dd;
scanf("%f", &fd);       // float型に入力
printf("%f¥n", fd);     // それを出力
scanf("%lf", &dd);      // double型に入力(%lfに注意)
printf("%f¥n", dd);     // それを出力(%fでよい)
```

printf()のときはfloat型の引数値が、double型の値に変換されて処理されるので、どちらの場合も唯一の変換指定子である"f"で対応できます。一方、scanf()の場合は、格納先がfloat型か、double型かを明確に識別するために、変換指定子の使い分けが必要になります。

138 書式つき入力4

標準入出力関数10

scanfの書式指定例

書式指定文字列を確認する記述例を示します。各記述の右側に書かれているのは出力結果例です。ここでは次の変数が宣言されているとします。

```
char ch;
short sh;
int  a, b, c, id, n;
unsigned int ud;
long lng;
double db;
long double ldb;
char ss[80];
void *vp = ss;
```

注：scanf()関数を連続実行させると、前のscanf()関数の読み残しによって、次のscanf()関数の動作が影響を受けることがある。以下の参考例は、それぞれscanf()関数を単独実行させた結果を示している。

■例1：変換指定子の確認

リスト	入力／出力	コメント
`scanf("%d", &a);`	10	通常のd変換数値入力をする
`printf("%d¥n", a);`	10	
`scanf("%d%d", &a, &b);`	20 30	ふたつの数字をd変換で入力する
`printf("%d %d¥n", a, b);`	20 30	
`scanf("%i", &id);`	10	i変換
`printf("%d¥n", id);`	10	
`scanf("%x", &id);`	10	x変換
`printf("%d¥n", id);`	16	
`scanf("%o", &id);`	10	o変換
`printf("%d¥n", id);`	8	
`scanf("%u", &ud);`	4000000000	u変換(intでは表現できない数を入力)
`printf("%u¥n", ud);`	4000000000	
`strcpy(ss, "ABC");`		
`vp = ss;`		
`printf("vp=%p¥n", vp);`	vp=0012FE5C	ss[0]のアドレス
`printf("*vp=%c¥n", *(char *)vp);`	*vp=A	ss[0]の文字
`scanf("%p", &vp);`	12FE5D	ss[1]のアドレスを入力
`printf("vp=%p¥n", vp);`	vp=0012FE5D	入力された値
`printf("*vp=%c¥n", *(char *)vp);`	*vp=B	ss[1]の値になる
`scanf("%c", &ch);`	a	1文字を変数に入れる(空白も入力可)
`printf("%c¥n", ch);`	a	
`scanf("%c", ss);`	b	1文字を配列に入れる
`printf("%c¥n", ss[0]);`	b	
`scanf("%s%n", ss, &n);`	abcde fgh	%nまでに入力された文字数
`printf("ss=%s n=%d¥n", ss, n);`	ss=abcde n=5	5個だった
`scanf(" %%%s",ss);`	%abcde	'%'をひとつ読み捨てる
`printf("%s¥n", ss);`	abcde	

`scanf("%[abcde]", ss);`	debug	a～eまでの文字を読む
`printf("%s¥n", ss);`	deb	
`scanf("%[a-e]", ss);`	debug	[a-e]と記述できる処理系が多い
`printf("%s¥n", ss);`	deb	
`scanf("%[^abcde]", ss);`	fortran	a～eでない文字を読む
`printf("%s¥n", ss);`	fortr	
`scanf("%[ a-z]", ss);`	test data123	スペース文字も読む
`printf("%s¥n", ss);`	test data	

■例2：長さ修飾子（省略可能）

`scanf("%hd", &sh);`	1234	short int変換
`printf("%hd¥n", sh);`	1234	
`scanf("%ld", &lng);`	123456789	long int変換
`printf("%ld¥n", lng);`	123456789	
`scanf("%lf", &db);`	1234567.1234	double変換
`printf("%f¥n", db);`	1234567.123400	
`scanf("%Le", &ldb);`	2.345678e+300	long double変換
`printf("%Le¥n", ldb);`	2.345678e+300	

■例3：フィールド幅（省略可能）

`scanf("%4d%3d%4s", &a, &b, ss);`	1234567890ab	4, 3, 4文字で区切る
`printf("a=%d b=%d ss=%s¥n", a, b, ss);`	a=1234 b=567 ss=890a	
`scanf("%4d%3d%4s", &a, &b, ss);`	123 45678 90	空白でも区切りになる
`printf("a=%d b=%d ss=%s¥n", a, b, ss);`	a=123 b=456 ss=78	
`strcpy(ss, "12345678");`		
`scanf("%5c", ss);`	AB CDEFG	5字読む。空白も読む。'¥0'付加せず
`printf("ss=%s¥n", ss);`	ss=AB CD678	"678"が残っている

■例4：代入抑止（省略可能）

`scanf("%d%*d%d", &a, &b);`	12 34 56	2番目の10進数を読み捨て
`printf("a=%d b=%d¥n", a, b);`	a=12 b=56	

■例5：入力の区切り

標準では空白が区切りとなる

`scanf("%d%d%d", &a, &b, &c);`	12 34 56	空白で入力区切り
`printf("a=%d b=%d c=%d¥n", a, b, c);`	a=12 b=34 c=56	

区切り文字として','を指定

`scanf("%d , %d , %d", &a, &b, &c);`	12 , 34,56
`printf("a=%d b=%d c=%d¥n", a, b, c);`	a=12 b=34 c=56

その他の区切り文字を指定

`scanf("%d / %d / %d", &a, &b, &c);`	12 / 34/56
`printf("a=%d b=%d c=%d¥n", a, b, c);`	a=12 b=34 c=56
`scanf("%d and %d and %d", &a, &b, &c);`	12 and 34and56
`printf("a=%d b=%d c=%d¥n", a, b, c);`	a=12 b=34 c=56

139 書式つき入力5

標準入出力関数11

scanf利用上の注意点

scanf()は癖のある関数で、ときには奇妙な(と思える)動作をします。ここでいくつか特徴的なことを説明します。

①%sで空白は読み込まない

scanf()の"%s"変換では空白がくると読み込みを終わります。ですから"this is string"といった文字列は読み込むことができません。空白をふくむ文字列を読み込むときはgets()やfgets()を使うのが便利です。

②%cなら空白も読む

%cを使うと空白も改行文字もすべて1文字読み込みします。%5cでは5文字を読み込みます。

③読み残しの処理

scanf()では読み残しがでることがあります。その中で「改行文字が残る」ことはよく発生します。事情を知らないと、おかしな動作に思えることがあります。次に例を示します。

```
int  nbr, ch1, ch2;
printf("数値入力:"); scanf("%d", &nbr);      // 10進数を読み込む
printf("数値=%d¥n", nbr);                    // それを出力
printf("文字入力1:"); ch1 = getchar();       // 1文字読み込む
printf("文字1=[%c]¥n", ch1);                 // それを出力
printf("文字入力2:"); ch2 = getchar();       // もう1文字を読み込む
printf("文字2=[%c]¥n", ch2);                 // それを出力
```

実行例

```
数値入力:1234          ――(1)数値入力
数値=1234              ――(2)それを出力
文字入力1:文字1=[      ――(3)1文字目を入力しようとしたら勝手に改行出力
]
文字入力2:a            ――(4)2文字目は入力できる
文字2=[a]              ――(5)その文字を出力
```

scanf()関数実行に対して(1)で正しく入力しても、最後の改行文字はストリームに残ります。(3)はその改行文字を読み込み、それを出力しています。これは当然のことですが、慣れないと関数が機能していないように見えます。この場合は、プログラマが適切に改行文字を読み捨てる必要があります。

■④読み込み不一致の処理

変換指定で期待していなかった入力データに出合うと、scanf()関数は読み込みを中止します。そのときストリームの文字はそのまま残ります。読み込み中止に対して無対策だと次の現象がおきます。

```
int n = 1234;
scanf("%d", &n);          // 入力：ABCD
printf("n=%d¥n", n);      // 出力：1234
```

数値入力の指示に対して、"ABCD"の入力は不適切なので、scanf()関数は読み込みを中止します。そのため変数nの元の値を表示しています。正しく実行されるかどうかをガードするには、たとえば次のようにします。

```
if (scanf("%d", &n) != 1) {        // 正しい入力なら戻り値は1
    // 不正入力対策を記述
}
```

第17章

C Quick Reference

ファイル処理関数

140 ファイル処理の手順1

ファイル処理関数1

簡単なファイル入出力

この章では代表的なファイル処理関数を示しながら、ファイル処理の方法を説明します。各関数の正確な仕様は「第19章 標準ライブラリの要約」(p. 247)を見てください。

ファイルとの間でデータの入出力を行なう場合、いきなり読み書きすることはできません。次の手順が必要です。

(1)処理したいファイルをオープンする
(2)入出力を行なう
(3)ファイルをクローズする

次に簡単なファイル入出力のサンプルを示します。

ファイル処理プログラム例1

```
#include <stdio.h>
int main(void)
{
    FILE *fin, *fout;
    char ss[256];

    if ((fout=fopen("file1.txt", "w")) == NULL) {
                                        // (1) 出力用ファイルを開く
        printf("出力ファイルをオープンできません。¥n");
        return 1;
    }
    fputs("aaaa¥n", fout);               // (2) ファイルに書き込む
    fputs("bbbb¥n", fout);
    fclose(fout);                        // (3) ファイルを閉じる

    if ((fin=fopen("file1.txt", "r")) == NULL) {
                                        // (4) 入力用ファイルを開く
        printf("入力ファイルをオープンできません。¥n");
        return 1;
    }
    while (fgets(ss, 256, fin) != NULL) {// (5) ファイル終了まで1行入力
        printf("%s", ss);                // (6) それを表示
    }
    fclose(fin);                         // (7) ファイルを閉じる
    return 0;
}
```

実行結果

```
aaaa
bbbb
```

このプログラムはfile1.txtファイルに文字列を出力しています。次に同じファイルから文字列入力をしています。

まず(1)でfopen()関数を使ってfile1.txtを書き込み用("w")にオープンしています。オープンに成功すると「ストリームへのポインタ」(後述)がfoutに戻ります。オープンに失敗すると、戻り値がNULLになるので、エラー処理をします。

正しくオープンされたファイルに対しては(2)のように、fputs()関数で文字列出力できます。このfputs()関数は、

――foutで示すファイルに"aaaa¥n"を書き込む

という指示です。書き込みが終了したら(3)でファイルをクローズします。ファイルクローズを行なうと、ディスク上でファイルが確定します。

次に(4)でfopen()関数を使ってfile1.txtを今度は読み込み用("r")にオープンしています。オープンに失敗すると、戻り値がNULLになるので、エラー処理をします。

正しくオープンされたファイルに対しては(5)のように、fgets()関数で文字列読み込みできます。このfgets()関数は、

――finで示すファイルから最大256文字の1行文字列をssに読み込む

という指示です。データがなくなったらfgets()関数はNULLを返すので、while文でそれを確認しています。読み込みが終了したら(7)でファイルをクローズします。

141 ファイル処理の手順2

ファイル処理関数2

ファイルのオープンとクローズ

ファイルオープンと**ファイルクローズ**用の関数の形式は次のとおりです。ファイルがオープンされたままでプログラムが終了すると、そのファイルは自動的にクローズされます。それでも正しくfclose()関数を書く方が記述は安定します。

fopenの書式

```
FILE *fopen(char *filename, char *openmode)
```

説明	**filename**で示されるファイルを、**openmode**で示されるモードでオープンして、ストリームに結びつける
戻り値	正常オープン時：ストリームへのポインタ、エラー時：**NULL**

fcloseの書式

```
int fclose(FILE *stream)
```

説明	**stream**で示すファイルをクローズする
戻り値	正常時：0、エラー時：**EOF**

openmodeの指定1

モード	処理	ファイルがないとき	ファイルがあるとき
"r"	読み込み(read)	**NULL**を返す	正常処理
"w"	書き込み(write)	新規作成	前の内容は捨てる
"a"	追加書き込み(append)	新規作成	前の内容の後ろに追加する
"r+"	読み書き(更新)	**NULL**を返す	正常処理
"w+"	読み書き(更新)	新規作成	前の内容は捨てる
"a+"	追加のための読み書き	新規作成	前の内容の後ろに追加する

openmodeの指定2(バイナリモード)

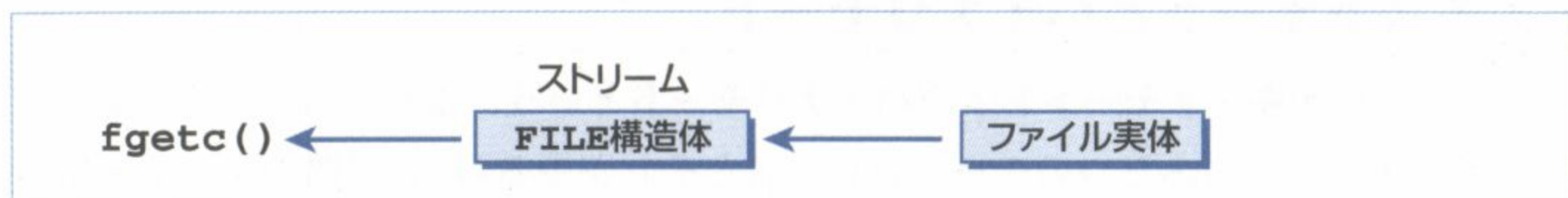
モード	処理方法
"rb", **"wb"**, **"ab"**	バイナリファイルに対する**r, w, a**処理を行なう
"r+b" または **"rb+"**	バイナリファイルに対する**r+**処理を行なう
"w+b" または **"wb+"**	バイナリファイルに対する**w+**処理を行なう
"a+b" または **"ab+"**	バイナリファイルに対する**a+**処理を行なう

注1：UNIXではテキストモードとバイナリモードを区別しないので"b"の付加は必要ない。

注2：更新モード(+つきモード)でリードからライトに、あるいはライトからリードに処理を替えるときはfflush()またはfseek(), fsetpos(), rewind()のいずれかの関数を使ってストリームを設定しなおす必要がある。

ストリーム

C言語ではファイル入出力を行なうとき、fgetc()などの入出力関数と、ファイル実体とが、直接やりとりするわけではありません。入出力データの処理を効率的に

行なうために、**「ストリーム」**という概念を使用します。実際にはFILE構造体で、ストリームを実現しています。

実際の入力動作では、まずファイル実体から、あらかじめ決められているサイズのデータを一度に、ストリームに転送します。そしてfgetc()関数はストリームから1文字ずつ取り出す、という動作をします。このようにストリームを用意することで、転送効率がよくなったり、入出力に関する便利な処理が可能になります。

ストリームの中には、使用しているバッファの位置や、バッファに残っている文字数や、次の読み書き位置などの情報が格納されています。

ファイルをオープン指示すると、そのファイル読み書きに必要な情報をFILE構造体に設定し、そのFILE構造体へのポインタを返します。以降は、そのポインタを指定することで、ファイル読み書きを行なうことができます。

```
FILE *fp;                         ――ポインタfpを宣言
fp = fopen("tst.txt", "r");       ――ファイルを開きストリームへのポインタをfpに返す
```

note ストリームと入出力単位

ワイド文字が導入されたことに合わせて、ストリームに**「入出力単位」**という概念が導入された。これはストリームをバイト用途か、ワイド用途かに切り替える機能である。

ストリームは**fopen()**関数などで、外部ファイルと結びつけられたとき**「入出力単位をもたない」**という状態になる。ここで最初にバイト文字入出力関数を実行すると**「バイト文字単位のストリーム」**になる。また最初にワイド文字入出力関数を実行すると**「ワイド文字単位のストリーム」**になる。さらに「入出力単位をもたない」という状態のときに**fwide()**関数を実行することで、どちらかに指定することもできる。

「バイト文字単位のストリーム」になっているとき、ワイド入出力関数を実行してはならない。逆も同じである。ストリームの入出力単位を変更するには、**freopen()**関数で再オープンする。この操作で「入出力単位をもたない」という初期状態に戻る。もちろんファイルをいったん終了して再度**fopen()**で開けば、新たに設定ができる。

なおワイド文字単位のストリームは、内部的に**mbstate_t**オブジェクトをもつ。このオブジェクトは、入出力ごとのワイド文字関連の変換状態情報(シフト状態など)を保持する。これらの事情は**stdin**, **stdout**, **stderr**でも同じであり、プログラム実行開始時には入出力単位をもたないという状態にある。

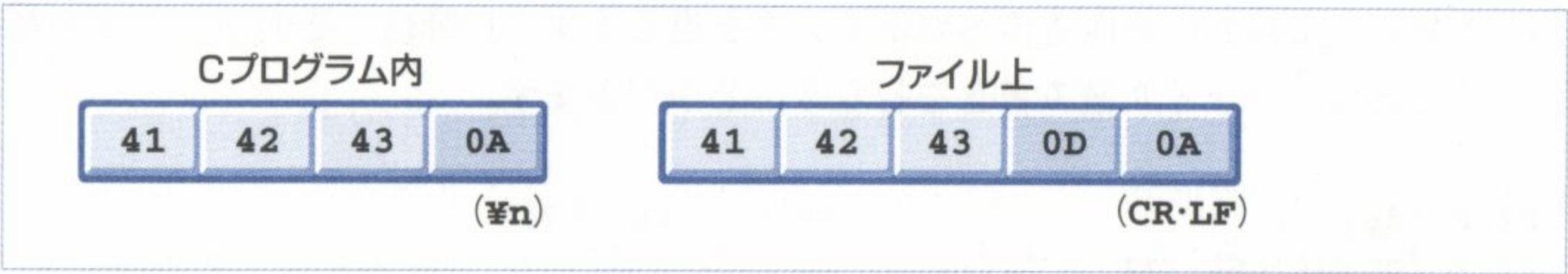
142 ファイル処理の手順3

ファイル処理関数3

テキストモードとバイナリモード

ファイルは**テキストモード**および**バイナリモード**という、ふたつのスタイルでオープンできます。このふたつのモードはC言語とそれが走るOSとの間のファイル処理の違いにより生じています。

たとえばWindowsではテキストの行末を0D(CR：復帰）と0A(LF：改行）というふたつの文字で表現しています。ディスク上のテキストデータはこのスタイルで保存されます。

一方、C言語は文字列の行末をおなじみの'¥n'(0A)で表現しています。たとえば"ABC¥n"という文字列は次のように格納されます。

この違いの整合をとるためには次の処理が行なわれなくてはなりません。この処理を自動的に行なってくれるのが「テキストモード」です。

ファイルに書き込むとき → ¥nをCR・LFに変換する
ファイルから読み込むとき → CR・LFを¥nに変換する

一方、この種の自動変換が困るときもあります。たとえばディスク上にあるファイル内容をそのままダンプ表示させるプログラムを作るときは、

——ディスク上の0D 0Aは画面上にもそのまま0D 0Aと表示しなければならない

という事情があります。このためディスク上のデータをそっくりそのまま読み込む機能も必要になります。すなわち、

CR・LF ↔ ¥nという自動変換を行なわない読み書き

というものも必要になります。それを行なうのが「バイナリモード」です。

バイナリモードを指定するときはファイルオープン時のモード指定に'b'の字を加えます。なおUNIXにおいてはファイルはテキスト形式もバイナリ形式も同一にあつかわれるのでこの区別は不要です。Cはそもそも UNIX上で使うように開発されたため、UNIXでのファイル管理方法とCでのデータ管理方法は同一になっています。もしUNIX環境でファイルのオープンモードで"rb"を指定しても、"r"と同じになります。

143 標準ストリーム

ファイル処理関数4

オープン済みのストリーム

Cではプログラムを実行するとき、特別に次の3つのファイルが自動的にオープンされています。これを**標準ストリーム**といいます。**stdin**と**stdout**は、通常のリダイレクト操作('<'と'>')の対象になります。

標準ストリーム名	機能	接続先
stdin	標準入力	通常はキーボード
stdout	標準出力	通常はディスプレイ
stderr	標準エラー出力	通常はディスプレイ

コンソールと入出力をする関数の入出力先は、次のように結びついています。

```
getchar() gets() scanf()  → stdinから入力
putchar() puts() printf() → stdoutに出力
```

したがってこれらの関数を使うと「129 標準入力と標準出力」(p. 182)で示したように、リダイレクト操作で接続先を切り替えることができます。

また通常のファイル入出力関数で、入出力先にstdin, stdout(またはstderr)を記述することもできます。次の例はどちらもキーボードから1文字入力をし、ディスプレイに1文字出力をします。

```
int ch;
ch = getchar();         // 標準入力先から1文字読む
ch = fgetc(stdin);      // 同上。同じ動作になる
putchar('A');           // 標準出力先に'A'を出力
fputc('A', stdout);     // 同上。同じ動作になる
```

画面固定の出力先

エラーメッセージなどのように、「リダイレクト操作しているときも常に画面に表示された方が便利」という出力があります。そのような用途のために標準エラー出力を示すstderrが用意されています。たとえば、

```
fprintf(stderr, "%sをオープンできません。¥n", filename);
```

という記述を行なうと、プログラム実行時に通常のリダイレクト指示('>')があっても、常にディスプレイに表示されます。

144 ファイル入出力関数1

ファイル処理関数5

ファイル入出力関数

オープンされたファイルとの間でデータの読み書きを行なうときは、専用の関数を使います。一般的なファイル入出力用の関数は次のとおりです。ここで各引数は、

```
int c;
int n;
char s[256];
FILE *fi;  ――入力用に用いる
FILE *fo;  ――出力用に用いる
```

と宣言されているものとします。

関数	説明
fputc(c, fo)	文字**c**を**fo**で示すファイルに出力 戻り値：書いた文字**c**。エラー時は**EOF**
putc(c, fo)	**fputc**と同じ（マクロ版の場合がある）
fgetc(fi)	**fi**で示すファイルから1文字入力 戻り値：入力文字。エラーとファイル終了時は**EOF**
getc(fi)	**fgetc**と同じ（マクロ版の場合がある）
fputs(s, fo)	文字列**s**を**fo**で示すファイルに出力 戻り値：負でない値。エラー時は**EOF**
fgets(s, n, fi)	**fi**で示すファイルから最大**n**バイトの文字列を**s**に入力 戻り値：ポインタ**s**。エラーとファイル終了時は**NULL**
fprintf(fo, 書式, 引数並び)	**fo**で示すファイルに指定書式で出力 戻り値：転送バイト数。エラー時は負の値
fscanf(fi, 書式, 引数並び)	**fi**で示すファイルから指定書式で入力 戻り値：入力項目数。エラー時は**EOF**

注：#include <stdio.h>が必要。

これらの用法はすでに説明した、

```
putchar() getchar() puts() gets() printf() scanf()
```

とほとんど同じですが、次の違いがあります。

- ストリームポインタで相手ファイルを指定する
- **fgets()**は読み込む最大文字数を指定できる

たとえばfgets()を使って、

```
fgets(s, 256, fi);    ──ファイルから文字列を読み込む
```

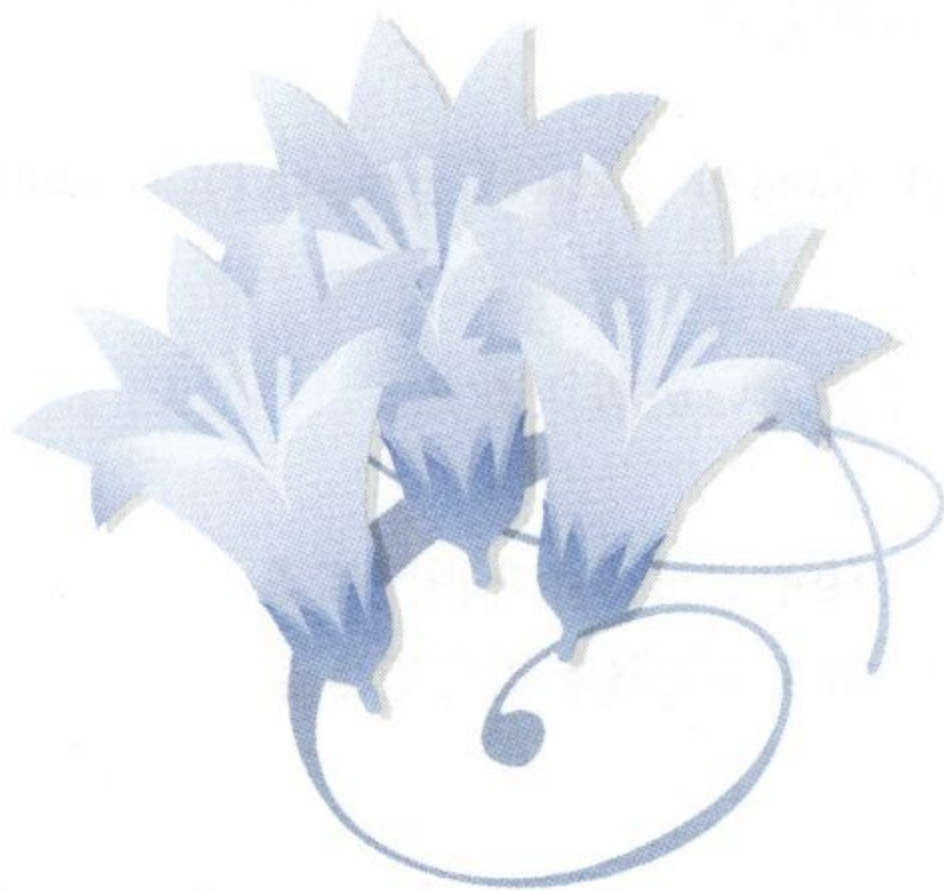
を実行するとfiで示すファイルから文字列を1行分だけsに読み込みます。そのとき文字列が256より長いときは('¥0'を含めて)256文字までを読み込みます。残りは次のfgets()などの入力処理で読まれます。この256文字というのは文字列の終端のコードである'¥0'をふくめています。したがって実際にファイルから読み込むのは255文字までです。

またputs()/gets()とfputs()/fgets()は改行のあつかいが次のように微妙に異なります。

- **puts()**は改行を付加する。**fputs()**は改行を付加しない。
- **gets()**は改行を除去する。**fgets()**は改行を残す(改行文字として読み込む)。

1文字読み込みと書き込みには、getc()/fgetc(), putc()/fputc()と、それぞれ2種類の関数があります。getc()関数とputc()関数は効率のためにマクロで実装されることがあります。

145 ファイル入出力関数2

ファイル処理関数6

ファイル入出力処理

ファイル入出力関数を用いて実際に処理を行なった例を示します。

ファイル処理プログラム例2

```
#include <stdio.h>
int main(void)
{
    FILE *fin, *fout;
    int ch, dt;
    char ss[256];

    printf("----fputc, fgetc¥n");
    if ((fout=fopen("file1.txt", "w")) == NULL) return 1;
    fputc('A', fout); fputc('B', fout);
    fclose(fout);

    if ((fin=fopen("file1.txt", "r")) == NULL) return 1;
    while ((ch=fgetc(fin)) != EOF) {
        putchar(ch);
    }
    fclose(fin);
    putchar('¥n');

    printf("----fputs, fgets¥n");
    if ((fout=fopen("file1.txt", "w")) == NULL) return 1;
    fputs("abcd¥n", fout); fputs("EFGH¥n", fout);
    fclose(fout);

    if ((fin=fopen("file1.txt", "r")) == NULL) return 1;
    while (fgets(ss, 256, fin) != NULL) {
        printf("%s", ss);
    }
    fclose(fin);

    printf("----fprintf, fscanf¥n");
    if ((fout=fopen("file1.txt", "w")) == NULL) return 1;
    fprintf(fout, "%d¥n", 1234); fprintf(fout, "%d¥n", 5678);
    fclose(fout);

    if ((fin=fopen("file1.txt", "r")) == NULL) return 1;
    while (fscanf(fin, "%d", &dt) == 1) {
        printf("%d¥n", dt);
    }
    fclose(fin);
    return 0;
}
```

実行結果

```
----fputc, fgetc
AB
----fputs, fgets
abcd
EFGH
----fprintf, fscanf
1234
5678
```

146 ブロック単位の読み書き ファイル処理関数7

freadとfwrite

fread()関数とfwrite()関数を使うと、**ブロック単位**(指定したサイズ単位)の入出力を行なうことができます。

freadの書式

```
size_t fread(void *buf, size_t size, size_t n, FILE *stream)
```

説明 指定のストリームから「sizeバイト×n個」分のデータをbufに読み込む
戻り値 読み込んだデータの個数を返す

fwriteの書式

```
size_t fwrite(void *buf, size_t size, size_t n, FILE *stream)
```

説明 bufの先頭から「sizeバイト×n個」分のデータを指定のストリームに書き込む
戻り値 書き込んだデータの個数を返す

ブロック読み書き処理

両関数の基本的な機能を確認するプログラムを示します。

ブロック単位の読み書きをするプログラム

```
#include <stdio.h>

typedef struct CItyp {
    char s[20];
    int  n;
} CItyp;

int main(void)
{
    FILE *fp;
    CItyp wk, dt[3] = {{"aaa", 100}, {"bbb", 200}, {"ccc", 300}};
    int ct, size = sizeof(CItyp);

    if ((fp = fopen("tst.txt", "w")) == NULL) return 1;
    for (ct=0; ct<3; ct++)
        fwrite(&dt[ct], size, 1, fp);      // 構造体1個分を書き込む
    fclose(fp);

    if ((fp = fopen("tst.txt", "r")) == NULL) return 1;
    while (1) {
        fread(&wk, size, 1, fp);           // 構造体1個分を読み込む
        if (feof(fp)) break;
        printf("s:%s n:%d¥n", wk.s, wk.n);
    }
    rewind(fp);
    return 0;
}
```

実行結果

```
s:aaa n:100
s:bbb n:200
s:ccc n:300
```

正しく読み込まれている

147 読み書き位置の指定1　ファイル処理関数8

fseekとftellとrewind

通常のファイル処理では、ファイルの先頭から最後まで順番に読み込む、あるいはファイルに順番に書き込みます。これは**シーケンシャルファイル処理**と呼ばれるものです。

Cには任意の位置を指定して読み書きを行なう機能もあります。

ストリームは、現在の読み書き位置を管理する「ファイル位置表示子」をもっています。fseek()関数、ftell()関数、rewind()関数はそのファイル位置表示子を調整します。これらの関数を使用すると、ファイル内の任意の位置とのアクセスを行なうことができます。

fseekの書式

```
int fseek(FILE *stream, long offset, int whence)
```

説明　指定ストリームのファイル位置表示子を**whence**を基点として**offset**バイト目の位置に移動する

戻り値　正常時：0、エラー時：非0

移動の基点（whence）の指定方法

名前	説明
SEEK_SET	先頭位置から
SEEK_CUR	現在位置から
SEEK_END	終端位置から

ftellの書式

```
long int ftell(FILE *stream);
```

説明　指定ストリームの現在のファイル位置表示子を返す

戻り値　正常時：現在のファイル位置表示子。エラー時：**-1L**
この戻り値は**fseek()**関数の引数として用いることができる

rewindの書式

```
void rewind(FILE *stream);
```

説明　指定ストリームのファイル位置表示子を先頭（0）にする
ファイル位置表示子移動に関しては**fseek(fp, 0L, SEEK_SET);**と同じ

fseek()関数はファイルのオープンモードによって指定方法が制約されます。自由に位置設定を行なうにはバイナリモードが適しています。

バイナリモード時の使用

whence として **SEEK_SET**, **SEEK_CUR**, **SEEK_END** を使用できる。**offset** は任意の値を書くことができる。

テキストモード時の使用

CR・LF↔**¥n** 変換があるので正しい移動ができないことがある。次の3つの方法は安全に指定できる。

(1) **whence** に **SEEK_SET**, **offset** に0を指定してファイル先頭に移動する
(2) **whence** に **SEEK_END**, **offset** に0を指定してファイル終端に移動する
(3) **whence** に **SEEK_SET**, **offset** に以前に実行した **ftell()** 関数の戻り値を指定して前回の位置に移動する

fseek() 関数の指定例を次に示します。offset 値は long 値なので L をつけています。「現在位置」とは、次に読み書き予定の位置のことをいいます。

```
fseek(fp, 0L,    SEEK_SET)    // 先頭に移動
fseek(fp, 10L,   SEEK_SET)    // 先頭から10文字目に移動
fseek(fp, 0L,    SEEK_END)    // 終端に移動
fseek(fp, -5L,   SEEK_END)    // 終端から5文字前に移動
fseek(fp, 10L,   SEEK_CUR)    // 現在位置から10文字後ろに移動
fseek(fp, -10L,  SEEK_CUR)    // 現在位置から10文字前に移動
```

fseek() 関数、ftell() 関数、rewind() 関数の利用例を示します。

fseek関数とftell関数の働き

```
#include <stdio.h>
void outch(char *p, int n)
{
    while (n--)
        putchar(*p++);
    putchar('¥n');
}

int main(void)
{
    FILE *fp;
    char ss[100];
    long int pos;

    fp = fopen("tst.txt", "wb");
    if (fp == NULL) return 1;
    fprintf(fp, "ABCDEFGHIJKLMNOPQRSTUVWXYZ");      // 最初のデータ設定
    fclose(fp);                                      // いったん閉じる

    puts("----オープン直後, 26文字読む");
    fp = fopen("tst.txt", "rb+");   // 読み書きするので "rb+" でオープン
```

```
    if (fp == NULL) return 1;
    fscanf(fp, "%26c", ss);
    outch(ss, 26);

    puts("----先頭から3文字目指定(先頭は0), 5文字読む");
    fseek(fp, 3L, SEEK_SET);
    fscanf(fp, "%5c", ss);
    outch(ss, 5);

    puts("----ftellの位置に戻す");
    pos = ftell(fp);                    // 次の'I'の位置を記憶
    fscanf(fp, "%5c", ss);
    outch(ss, 5);
    fseek(fp, pos, SEEK_SET);           // 'I'の位置に戻す
    fscanf(fp, "%5c", ss);
    outch(ss, 5);

    puts("----終端から10文字前を指定, 5文字読む");
    fseek(fp, -10L, SEEK_END);
    fscanf(fp, "%5c", ss);
    outch(ss, 5);

    puts("----先頭から10文字目を指定, abcdeを書く");
    fseek(fp, 10L, SEEK_SET);
    fprintf(fp, "%s", "abcde");

    puts("----先頭から26文字を読む");
    fseek(fp, 0L, SEEK_SET);
    fscanf(fp, "%26c", ss);
    outch(ss, 26);

    puts("----rewindをして先頭から26文字を読む");
    fseek(fp, 0L, SEEK_SET);
    fscanf(fp, "%26c", ss);
    outch(ss, 26);
    fclose(fp);
    return 0;
}
```

実行結果

```
----オープン直後, 26文字読む
ABCDEFGHIJKLMNOPQRSTUVWXYZ
----先頭から3文字目指定(先頭は0), 5文字読む
DEFGH
----ftellの位置に戻す
IJKLM
IJKLM
----終端から10文字前を指定, 5文字読む
QRSTU
----先頭から10文字目を指定, abcdeを書く
----先頭から26文字を読む
ABCDEFGHIJabcdePQRSTUVWXYZ
----rewindをして先頭から26文字を読む
ABCDEFGHIJabcdePQRSTUVWXYZ
```

148 読み書き位置の指定2

ファイル処理関数9

fgetposとfsetpos

読み書き位置の移動はfgetpos()関数とfsetpos()関数で行なうこともできます。fseek()関数とftell()関数は、読み書き位置をlong型で管理します。一方、fgetpos()とfsetpos()はfpos_t型で管理するので、より大きなファイルをあつかうことができます。たとえばfpos_t型の定義例は「typedef long long fpos_t;」のようになります。

fgetposの書式

```
int fgetpos(FILE *stream, fpos_t *pos);
```

説明　指定ストリームの現在のファイル位置表示子などの情報を***pos**に取得する
戻り値　正常時：0、エラー時：非0

fsetposの書式

```
int fsetpos(FILE *stream, const fpos_t *pos);
```

説明　指定ストリームのファイル位置表示子を***pos**でセットしなおす
戻り値　正常時：0、エラー時：非0

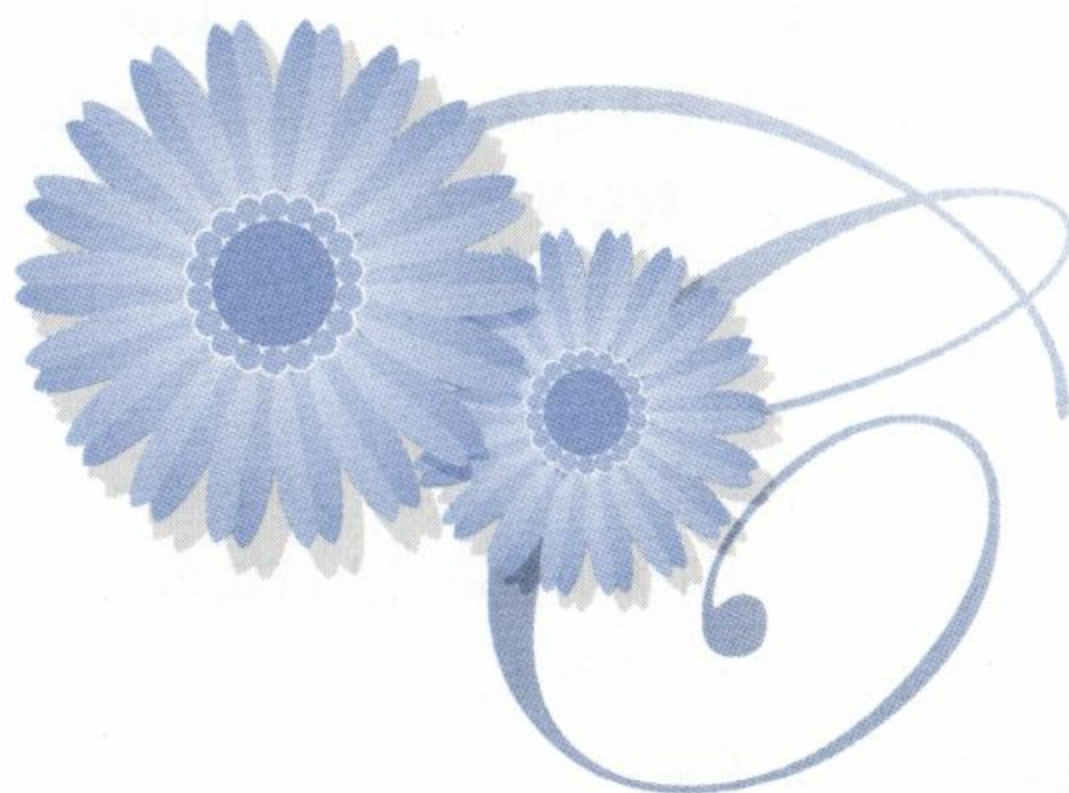
fgetpos()関数とfsetpos()関数の利用例を示します。

fgetpos関数とfsetpos関数の働き

```
#include <stdio.h>
int main(void)
{
    FILE *fp;
    int i, ch;
    fpos_t pos;                      // 位置保存用変数

    if ((fp=fopen("tst.txt", "w")) == NULL) return 1;
    fputs("ABCDEFGHIJKLMN", fp);     // 書込み
    fclose(fp);

    if ((fp=fopen("tst.txt", "r")) == NULL) return 1;
    for (i=1; i<=4; i++)
        putchar(fgetc(fp));          // 4文字読込み
    putchar(':');                    // 目印表示
    fgetpos(fp, &pos);               // 4文字読込み後の位置を取得
    while ((ch=fgetc(fp)) != EOF)
        putchar(ch);                 // 残りを出力
    putchar('¥n');
```

```
    fsetpos(fp, &pos);                   // ファイルの位置を復元
    while ((ch=fgetc(fp)) != EOF)       // 復元位置から読込み
        putchar(ch);
    putchar('¥n');

    fclose(fp);
    return 0;
}
```

実行結果

```
ABCD:EFGHIJKLMN
EFGHIJKLMN   ——位置が復元されている
```

149 ファイルエラー処理1　ファイル処理関数10

Cのファイル入力用関数はそれぞれにファイルが終了したことを戻り値で知る機能をもっています。たとえばfgetc()関数ではEOF値が該当します。一方これとは別に、ファイルの終了を知る関数も用意されています。またファイル処理時にエラーが発生したことを知る関数も用意されています。ファイル終了、およびファイルエラー検出に関連する関数は次のとおりです。

関数	説明
feof(fp)	ファイル終了なら真を返す
ferror(fp)	ファイルエラーなら真を返す
perror(文字列)	**stderr**に「文字列」とシステムエラーメッセージを出力
strerror(番号)	番号に対応したシステムエラーメッセージ文字列を得る
clearerr(fp)	ファイル終了表示子とエラー表示子をクリアする

注1：fpはストリームへのポインタ。

注2：strerrorは#include <string.h>が必要。他は#include <stdio.h>が必要。

feofとferror

fgetc()関数は規格では、(1)ファイルの終わりに達したとき、(2)ファイル読み取りエラーが生じたとき、にEOFを返します。またfgets()関数の戻り値NULLも「ファイル終了またはエラー発生」を意味しています。

どちらの場合も読み込み続行はできないので、通常はおなじみの、

```
while ((ch=fgetc(fin)) != EOF) { ... }
```

のような記述をすれば十分です。しかし(1)と(2)を区別して処理したいときは、feof()関数とferror()関数を使います。

またfread()関数は「*n*個のデータを読み込め」という指示を出し、その戻り値は「正しく読み込まれたデータ個数」です。もし指示した個数と戻り値の値が異なっていたら、「最後の半端な数のデータ読み込み、またはエラー発生」という重複情報になります。この場合は、feof()関数を使わないと、ファイル終了を正しく判定できません。

ファイルの読み込み時にファイル終了になると、ストリームのもっているファイル終了表示子がセットされます。ファイルの読み書き時にエラーが生じると、ストリームのもっているエラー表示子がセットされます。feof(), ferror()の両関数はこれらのフラグの状態を検査しています。

次にfgetc()を使った場合の、エラー処理例を示します。

feofを使うプログラム

```
#include <stdio.h>

int main(void)
{
    FILE *fp;
    int ch;
    if ((fp=fopen("tst.txt", "r")) == NULL) return 1;
    while (1) {
        ch = fgetc(fp);
        if (feof(fp)) break;                         // 終了確認
        if (ferror(fp)) {
            puts("読み込みエラーです。"); return 1;    // エラー確認
        }
        putchar(ch);
    }
    fclose(fp);
    return 0;
}
```

150 ファイルエラー処理2

ファイル処理関数11

clearerr

feof()関数が真になると、以後は何度feof()関数を実行しても真になります。ferror()関数も同じです。これはファイル終了表示子と、エラー表示子がセットされたまま維持されているからです。

この両表示子をクリアするには、

──ファイルを閉じるか、clearerr()関数を実行

します。clearerr()関数は次のように使います。

```
clearerr(fp); ──fpの指すストリームのファイル終了表示子とエラー表示子をクリア
```

またファイル終了表示子の場合は、「次の読み書き位置がファイル終端である」という状況を解除することでも、解消されます。具体的には次のような関数が、ファイル終了表示子をクリアします。

```
fopen()  freopen()  ungetc()  rewind()  fseek()
fsetpos()  clearerr()
```

perrorとstrerror

Cは処理系定義のエラーが発生すると、そのエラー番号を、システムがもっている、

errno

と呼ぶ左辺値(マクロまたは識別子で実現)に格納します。perror()関数はこのerrnoに対応するエラーメッセージを標準エラー出力(stderr)に表示します。errnoに関する定義はerrno.hファイルに入っています。

perror()関数の記述例を示します。

```
perror("異常発生");    // (1)
perror(NULL);          // (2)
```

(1)では、「異常発生」というメッセージと、現在のerrno対応のメッセージが表示されます。(2)ではerrno対応のメッセージだけが表示されます。

strerror()関数も同じようにerrnoとエラーメッセージを対応付けるものですが、この関数はエラーメッセージ文字列へのポインタを返します。次に記述例を示します。

perror関数とstrerror関数の使用例

```
#include <stdio.h>
#include <string.h>  /* strerror() */
#include <errno.h>

int main(void)
{
    FILE *fp;

    printf("errno=%d¥n", errno);        // 起動時のエラー番号
    if ((fp=fopen("noexist.txt", "r")) == NULL) {
                                        // 非存在ファイルを開く
        perror("不正な処理です");
        printf("%s¥n", strerror(errno));
        printf("errno=%d¥n", errno);    // エラー番号を表示
        return 1;
    }
    fclose(fp);

    return 0;
}
```

実行結果：Visual C++で実行

```
errno=0                                     ――最初の値は0(エラーではない)
不正な処理です: No such file or directory  ――perror
No such file or directory                   ――strerror
errno=2                                     ――エラーが発生している
```

第18章

C Quick Reference

その他の処理関数

151 文字処理

その他の処理関数1

文字処理用関数として次のものがあります。文字処理関数は1文字単位でテストまたは変換を行ないます。これらの関数はマクロで実現されている場合もあります。引数と戻り値はいずれもint型をとります。

またロケールを、デフォルトの"C"でなく特別のものに設定しているときは、その文化圏固有の動作になることがあります。以下に示すのは"C"ロケールの場合です。

関数	説明
isalnum(c)	英数字なら真
isalpha(c)	英文字なら真
iscntrl(c)	制御文字なら真
isdigit(c)	10進数字なら真
isgraph(c)	表示可能文字なら真
islower(c)	小文字なら真
isprint(c)	空白以外の表示可能文字なら真
ispunct(c)	区切り文字なら真
isspace(c)	空白類文字なら真
isupper(c)	大文字なら真
isxdigit(c)	16進文字なら真
tolower(c)	文字cを小文字に変換
toupper(c)	文字cを大文字に変換

注：#include <ctype.h>が必要。

注：変数はint c;。

利用例を次に示します。

```
#include <stdio.h>
#include <ctype.h>
int c1, c2;
...
if (isalpha(c1)) putchar(c1);      ──もしc1が英文字ならそれを出力
c2 = tolower(c1);                  ──文字c1を小文字にして変数c2に入れる
```

152 文字列処理

その他の処理関数2

主要な文字列処理用関数として次のものがあります。文字列処理関数は文字列の設定、比較、検索などを行ないます。

関数	説明
strlen(st)	文字列stの長さを返す
strcpy(ss, st)	配列ssに文字列stをコピーする
strcat(ss, st)	配列ssの後ろに文字列stを連結する
strcmp(s1, s2)	文字列s1と文字列s2を比較し以下の値を返す s1>s2 なら 戻り値は正(>0) s1<s2 なら 戻り値は負(<0) s1==s2 なら 戻り値は0(==0)
strncpy(ss, st, n)	配列ssに文字列stの先頭n文字をコピーする
strncat(ss, st, n)	配列ssの後ろに文字列stの先頭n文字を連結する
strncmp(s1, s2, n)	文字列s1と文字列s2の先頭n文字同士を比較する
strchr(st, ch)	文字列stの中の文字chの位置を返す。ないときはNULLを返す
strstr(s1, s2)	文字列s1の中の文字列s2の位置を返す。ないときはNULLを返す

注：#include <string.h>が必要。

注：ssは文字列変数。st, s1, s2は文字列変数または文字列リテラル。nは整数。

文字列処理関数の簡単な使用例を次に示します。

```
#include <stdio.h>
#include <string.h>
...
char a[80], b[80], *p;
int n;

n = strlen(a);                  // 文字列aの長さをnに入れる

strcpy(a, "abcde");             // 文字列aに文字列"abcde"をコピーする
strcpy(b, "12345");             // 文字列bに文字列"12345"をコピーする
strncpy(a, b, 3);               // 先頭3文字をコピー。aは"123de"になる
a[3] = '¥0';                    // aは"123"になる

strcat(a, b);                   // 文字列aに文字列bを連結する
strncat(a, b, 4);               // 文字列aに文字列bの先頭4文字を連結する

if (strcmp(a, b) > 0)           // 文字列aとbを比較する
    printf("左辺が大¥n");

if (strncmp(a, b, 5) > 0)       // 文字列aとbの先頭5文字ずつを比較する
    printf("左辺が大¥n");

strcpy(a, "Fortran and Pascal");
p = strchr(a, 'P');             // 文字列aの中から文字'P'の位置を探す
```

```
if (p != NULL) puts(p);     // 出力："Pascal"になる
p = strstr(a, "and");       // 文字列aの中から文字列"and"の位置を探す
if (p != NULL) puts(p);     // 出力："and Pascal"になる
```

なおWindowsのように日本語表現にシフトJISコードを使用している環境では、strchr()とstrstr()関数は漢字非対応なので注意してください。2バイトで構成される漢字コードの2バイト目に、検索文字(列)があると処理がおかしくなります。

次に例を示します。この例で漢字「海」のコードは8A 43です。2バイト目が文字'C'のコードと一致するため、それが検出されています。

```
char *p, ss[] = "空と海ですABCDE";

p = strchr(ss, 'C');        ――文字'C'を探す
if (p != NULL) puts(p);     ――出力："CですABCDE"

p = strstr(ss, "C");        ――文字列"C"を探す
if (p != NULL) puts(p);     ――出力："CですABCDE"
```

153 メモリ管理

その他の処理関数3

実行時にヒープ領域と呼ばれるところから、メモリを確保したり、解放したりすることができます。それには次の関数を使います。これらの関数を用いることで動的メモリ確保ができます。

関数	説明
void *malloc(size_t size)	**size**バイトのメモリ確保
void *calloc(size_t n, size_t size)	**size**バイト×**n**個のメモリ確保
void *realloc(void *ptr, size_t newsize)	**ptr**で示す確保済みの領域を**newsize**で再確保
void free(void *p)	**ptr**で示すメモリ領域を解放

注：#include <stdlib.h>が必要。

malloc(), calloc(), realloc()の各関数の返す値はvoid型へのポインタです。この値は任意のポインタ変数に代入できます。また使いたいデータ型にキャストして明示的にするのもよいスタイルです。C++との互換性を考慮すればキャストするのが無難です。

malloc()関数は指定バイトのメモリ確保を行ないます。calloc()関数は「基本サイズ×n個」という確保サイズ指定を行ないます。calloc()関数で確保した領域は「0で初期化される」というサービスがあります。

realloc()は確保済みの領域のサイズを変更したいときに使用します。

これらの関数で確保された領域は、適切に境界調整されています。

確保サイズとして0を指定したときの動作は処理系定義です。

free()関数は、確保済みの領域を解放します。ただし確保したメモリ領域をプログラムの終了時点まで使用しているときは、わざわざfree()関数でメモリ解放することはありません。そのプログラムが終了すると、使用していたメモリ領域はOSにより自動的に解放されます。

malloc(), calloc(), realloc()の各関数は、メモリ確保できないときにNULLを返します。エラーチェックするときは次のように記述します。

```
char *p;
p = (char *)malloc(1000);
if (p == NULL) {                            // メモリ確保に失敗したら
    printf("メモリ確保ができません¥n");      // メッセージを出して
    exit(1);                                // 処理を打ち切る
}
```

次にいろいろなメモリ確保例を示します。ここではエラーチェックは省略します。

■①malloc関数でメモリ確保し解放する

```
char *p;
p = (char *)malloc(1000);          // 1000バイト確保
// 仕事をする
free(p);                           // 解放する
```

■②calloc関数で「int型サイズ×500個分」をメモリ確保する

```
int *p;
p = (int *)calloc(500, sizeof(int));
```

■③realloc関数でメモリ再確保する

```
char *p, *tmp;
p = (char *)malloc(500);           // (1) 500バイト確保
strcpy(p, "abcd"); puts(p);        // (2) 出力：abcd
tmp = (char *)realloc(p, 1000);    // (3) 1000バイトに再設定
if (tmp != NULL) p = tmp;          // (4) 正しさのチェック
strcat(p,"ABCD"); puts(p);         // (5) 出力：abcdABCD
```

realloc()関数では、新しく確保された領域が、前の領域と同じ位置とは限りません。しかし前のデータは(新しいメモリサイズ内で)維持されます。(5)の出力でそれが確認できます。また(3)の記述は、

```
p = (char *)realloc(p, 1000);    // 左辺も引数もp
```

でも通常は問題ありませんが、メモリ再確保に失敗したとき、pに戻り値NULLが代入されるため、pが元もと指していたメモリ領域にアクセスできなくなります。(3)(4)の記述をすると安全です。

154 時間処理 1

その他の処理関数4

時間処理用関数

コンピュータは内部にシステムクロックをもっています。次に示す関数はシステムクロックを取得して処理するものです。これらの関数を使うと、待ち時間処理や、実行時間測定や、*n*秒ごとに何かを行なうインターバル処理などを行なうことができます。

関数	説明
time	現在の暦時刻（**time_t**型）を得る
localtime	暦時刻を構造体型の現地時刻に変換する
gmtime	暦時刻を構造体型の協定世界時（世界標準時）に変換する
asctime	構造体型の時刻を表示可能文字列に変換する
ctime	暦時刻を現地時刻の表示可能文字列に変換する
mktime	構造体型の時刻を暦時刻（**time_t**型）に変換する
difftime	ふたつの時刻の差を計算する
clock	プログラム実行開始からの経過時間を得る
strftime	時刻・日付情報の書式化して文字列にする

注：#include <time.h>が必要。

時間表現用データ型

これらの関数は時間を表現するために次に示す4つのデータ型を使います。「char *」以外の型はtime.hの中で用意されています。またdifftime()関数は戻り値としてdouble型を使います。時間処理用のデータ型を次に示します。

(1) **char *型** ——文字列表現用

(2) **clock_t型** ——通常は**typedef**で**long**型に定義されている

(3) **time_t型** ——通常は**typedef**で**long**型に定義されている

(4) **tm構造体型** ——少なくとも以下の情報をふくまなければならない

```
int tm_sec;      // 秒 [0～61] C99では0～60
int tm_min;      // 分 [0～59]
int tm_hour;     // 時間 [0～23]
int tm_mday;     // 日付 [1～31]
int tm_mon;      // 月 [0～11]
int tm_year;     // 1900年からの経過年数
int tm_wday;     // 曜日 [0：日 1：月 … 6：土]
int tm_yday;     // 1月1日から日数 [0～365]
int tm_isdst;    // 夏時間のフラグ
```

注：tm_secが59秒より大きいのは、うるう秒表現のためである。

時間処理の方法

clock()関数はプログラムがスタートしてからの経過時間を知らせてくれます。このとき戻り値は1秒以下の微小単位からなっています。ですからこれを秒単位に直すためにCLOCKS_PER_SECという記号定数が用意されています。clock()関数の戻り値をCLOCKS_PER_SECで割ることで秒数を知ることができます。clock()関数の戻り値は大きいのでclock_t型の変数を使います。

time()関数を使うとシステムの保有している時間情報を、time_t型の値として得ることができます。この値は単なる数値です。localtime()関数とgmtime()関数を使うと、time_t型値からtm構造体型にコンバートし、内容が理解可能になります。localtime()は日本時間を設定します。gmtime()は協定世界時を設定します。両者は時間が9時間ずれています。

さらにasctime()関数を使うと、tm構造体型データから、時刻を表示する文字列を生成します。現地時間文字列を取得するには「localtime() + asctime()」という記述が必要ですが、両者を合わせた機能をもつctime()関数も用意されています。

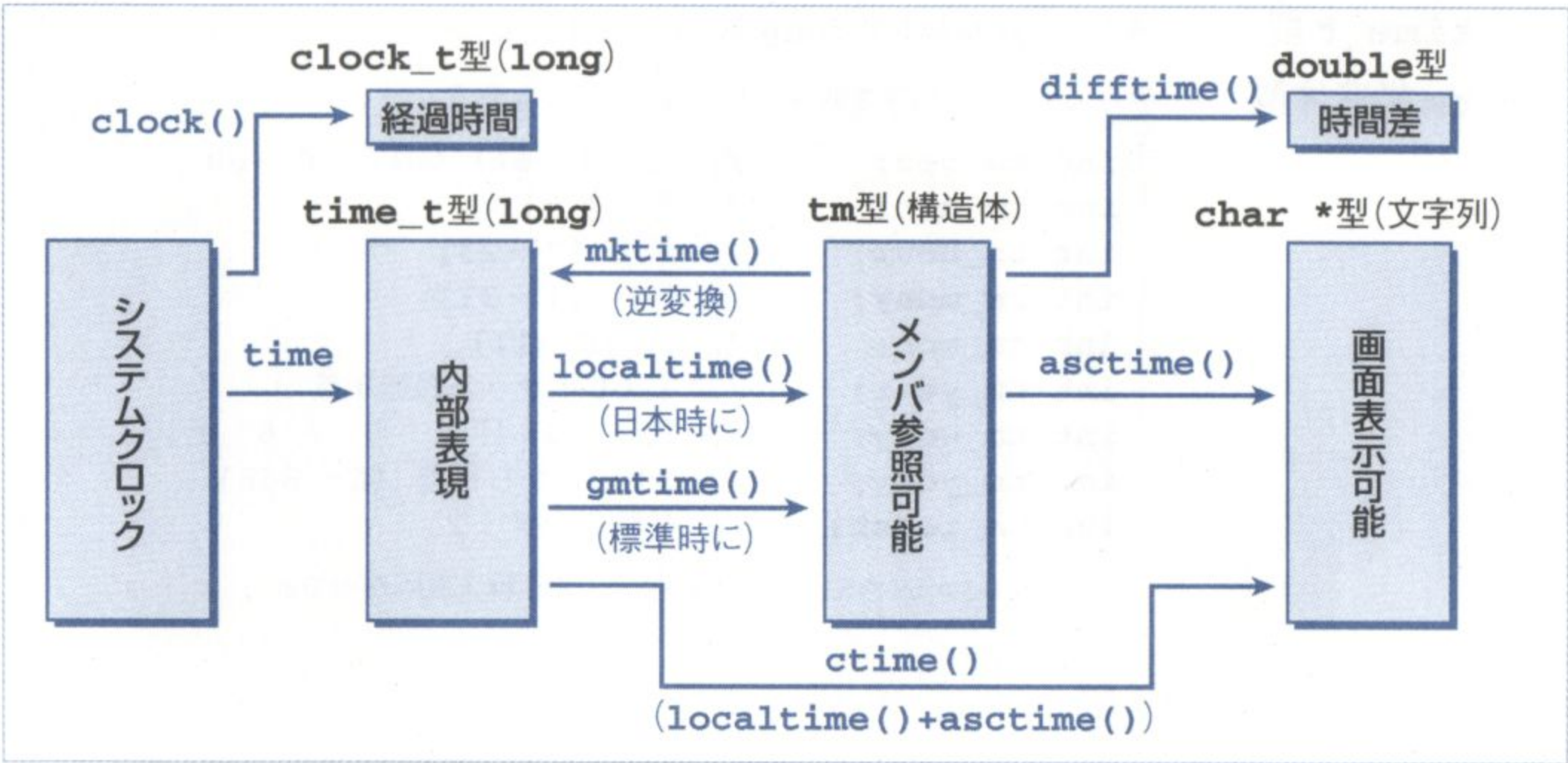
mktime()関数は逆にtm構造体型データからtime_t型データに変換するものです。

またstrftime()関数は、tm構造体の情報を元に、時刻・日付情報を、変換指定文字を使って書式化して文字列に格納する仕事をします。

関数の機能とデータの流れを次に示します。

時間処理関数によるデータの流れ

155 時間処理2

その他の処理関数5

時間差情報の提供

日本環境においては、gmtime() 関数が返す協定世界時と localtime() 関数が返す日本時には時間差（日本が9時間早い）があります。この時間差情報の提供方法は処理系に依存します。例として、Visual C++では地域時間差の設定を次のルールで行なっています。

> **localtime()** 関数は **TZ** 環境変数を使用する。**TZ** が設定されていない場合、OS が指定するタイムゾーン情報の使用を試行する。この情報が使用できない場合は既定値を使用する
> 例：Visual C++で環境変数設定を行なうときは"**set TZ=JST-9**"を実行

時間処理関数の簡単な利用例、およびtm構造体の利用例を示します。

現在時刻を表示する

```
#include <stdio.h>
#include <time.h>

int main(void)
{
    time_t mytime;
    struct tm *ltime;
    char *p;

    time(&mytime);                      // 暦時刻を得る
    ltime = gmtime(&mytime);            // 標準時間の構造体に変換
    p = asctime(ltime);                 // 表示可能文字列に変換
    printf("標準時間: %s", p);          // それを表示

    ltime = localtime(&mytime);         // 現地時間の構造体に変換
    p = asctime(ltime);                 // 表示可能文字列に変換
    printf("日本時間: %s", p);          // それを表示

    p = ctime(&mytime);                 // localtime()+asctime()の動作
    printf("ctime:    %s", p);          // それを表示
    return 0;
}
```

実行結果

```
標準時間: Wed Apr 28 02:30:15 2010
日本時間: Wed Apr 28 11:30:15 2010
ctime:    Wed Apr 28 11:30:15 2010
```

tm構造体の時間要素を出力する

```
#include <stdio.h>
#include <time.h>
int main(void)
{
    time_t mytime;
    struct tm *ltime;

    time(&mytime);                                // 暦時刻取得
    ltime = localtime(&mytime);                   // 現地時間に変換
    printf("月=%d¥n", ltime->tm_mon + 1);         // tm構造体メンバを出力
    printf("日=%d¥n", ltime->tm_mday);
    printf("時=%d¥n", ltime->tm_hour);
    printf("分=%d¥n", ltime->tm_min);
    printf("秒=%d¥n", ltime->tm_sec);
    printf("%s", asctime(ltime));                 // asctime()で出力
    return 0;
}
```

実行結果

```
月=4
日=28
時=11
分=32
秒=46
Wed Apr 28 11:32:46 2010
```

156 プログラム終了関数

その他の処理関数6

exit()関数を使うと、main()関数内だけでなく、任意の関数内でプログラムを終了することができます。次のように利用します。値0の代わりに定数EXIT_SUCCESS, 値1の代わりに定数EXIT_FAILUREを使うこともできます。これらの記号定数は、stdlib.hの中で定義されています。

```
exit(0);              ――正常終了
exit(1);              ――エラー時は1以上の値にするのが慣例
exit(EXIT_FAILURE);   ――定数での記述例。#include <stdlib.h>が必要
```

157 ワイド文字と多バイト文字の処理1 その他の処理関数7

関数	説明
mblen	多バイト文字1文字のバイト数を返す
mbtowc	多バイト文字をワイド文字に変換する
wctomb	ワイド文字を多バイト文字に変換する
mbstowcs	多バイト文字列をワ+イド文字列に変換する
wcstombs	ワイド文字列を多バイト文字列に変換する

注：#include <stdlib.h>が必要

ワイド文字と**多バイト文字**（マルチバイト文字：multibyte character）は多国語対応の機能です。ここではワイド文字と多バイト文字の基本処理についてまとめます。

ワイド文字と多バイト文字

1バイトでは表現できない文字セット（日本語や中国語など）を表現するために、C規格では多バイト文字、およびワイド文字という機能を用意しています。

多バイト文字は、1バイト文字（char型の文字）を連続させたもので、その実装は環境によります。たとえばWindowsではシフトJISコード、UNIXではEUCコード（Extended UNIX Code）で構成するのが通常です。

またJISコード（シフトJISコードではない）の場合は、「シフトイン文字」と「シフトアウト文字」を使って、通常文字（1バイト文字）と全角文字を区別します。シフトイン文字（JISコードでは0x0f）がくると、以降の文字は全角文字と判断し、シフトアウト文字（JISコードでは0x0e）がくると、通常文字解釈に戻ります。これを「シフト状態に依存した表現形式」といいます。

シフトイン文字 + 文字列 + シフトアウト文字

文字列の意味が通常とは変わる

一方、ワイド文字は1バイト文字および1バイトでは表現できない文字の全セット、つまりワイド文字セットを単一の整数型で表現するものです。その整数型としてwchar_t型を用意しています（stddef.h内で定義）。その仕様は処理系定義ですが、多くは、

```
typedef unsigned short wchar_t;
```

の定義を用いています。wchar_t型を使うと、通常の半角文字'A'も、全角文字の'漢'も、すべてひとつの整数型で表現できます。ワイド文字セットのコード体系についてはCでは規定がありませんが、代表例としてはUnicodeがあります。

ワイド文字は通常の関数用法で処理しようとすると不都合が生じます。たとえばワイド文字列を表示させたいときは、wprintf()関数を使うか、printf()関数+ワイド文字列変換指定を使います。本格的にワイド文字を活用するためには、ワイド文字(列)対応の関数群が必要になります。次にワイド文字と多バイト文字の簡単な処理例を示します。ワイド文字表示にprintf()+%ls変換指定を使用しています。

ワイド文字と多バイト文字の処理例

```
#include <stdio.h>
//#include <stdlib.h>
#include <locale.h>
#include <wchar.h>
int main(void)
{
    char ms[] = "多バイト文字列ABC";
    wchar_t ws[] = L"ワイド文字列ABC";
    setlocale(LC_ALL, "");          // ロケールをデフォルト(日本)にする
    printf("ms=%s¥n", ms);          // 出力: ms=多バイト文字列ABC
    printf("ws=%ls¥n", ws);         // 出力: ws=ワイド文字列ABC
    return 0;
}
```

158 ワイド文字と多バイト文字の処理2 その他の処理関数8

ワイド文字と多バイト文字の変換

ワイド文字と多バイト文字は相互に変換できます。この機能を使うと、ワイド文字(列)を標準の関数群で処理できて便利です。変換用関数としてはC89規格で次のものがあります。

```
mblen()      ――多バイト文字構成バイト数を返す
mbtowc()     ――多バイト文字をワイド文字に変換
wctomb()     ――ワイド文字を多バイト文字に変換
mbstowcs()   ――多バイト文字列をワイド文字列に変換
wcstombs()   ――ワイド文字列を多バイト文字列に変換
```

これらの処理は地域性に依存しますので、国別情報を設定することが必要です。その目的でsetlocale()関数が用意されています。基本用法は次のとおりです。地域性を日本にするときは(2)を使います。

```
setlocale(LC_ALL, "C" );   ――(1)最小環境にする(デフォルト)
setlocale(LC_ALL, "");     ――(2)文化圏に固有の地域性(日本)にする
```

多バイト文字を構成する最大バイト数は次の定数で知ることができます。

```
MB_CUR_MAX   ――現在のロケールにおいて多バイト文字を表現する最大バイト数
               (stdlib.h内で定義)
MB_LEN_MAX   ――処理系がサポートしているすべてのロケールにおいて、多バイト文字を
               表現するのに必要な最大バイト数(limits.hで定義)。MB_CUR_
               MAX <= MB_LEN_MAXになる
```

多バイト文字処理用の関数では、これらの定数を使うと記述が適切になります。たとえば多バイト文字の構成バイト数を返すmblen()関数は次のように記述できます。

```
n = mblen(mbs, MB_CUR_MAX);
  ――配列mbsの先頭からMB_CUR_MAX個の文字を検査して多バイト文字構成バイト数を返す
```

159 ワイド文字と多バイト文字の処理3 その他の処理関数9

ワイド文字処理用関数

ワイド文字処理用として用意されている主要な関数を次に示します。すべての関数については「第19章 標準ライブラリの要約」(p. 247)を見てください。

■ stdlib.hの関数

多バイト文字(列)・ワイド文字(列)変換関数

```
mblen() mbtowc() wctomb() mbstowcs() wcstombs()
```

■ wchar.hの関数

書式つきワイド文字入出力関数

```
fwprintf() fwscanf() swprintf() swscanf() vfwprintf() vfwscanf()
vswprintf() vswscanf() vwprintf() vwscanf() wprintf() wscanf()
```

ワイド文字入出力関数

```
fgetwc() fgetws() fputwc() fputws() fwide() getwc() getwchar()
putwc() putwchar() ungetwc()
```

ワイド文字列数値変換関数

```
wcstod() wcstof() wcstold() wcstol() wcstoll() wcstoul() wcstoull()
```

ワイド文字列コピー関数

```
wcscpy() wcsncpy() wmemcpy() wmemmove()
```

ワイド文字列連結関数

```
wcscat() wcsncat()
```

ワイド文字列比較関数

```
wcscmp() wcscoll() wcsncmp() wcsxfrm() wmemcmp()
```

ワイド文字列探索関数

```
wcschr() wcscspn() wcspbrk() wcsrchr() wcsspn() wcsstr() wcstok()
wmemchr()
```

長さ取得およびメモリセット関数

```
wcslen() wmemset()
```

ワイド文字時間変換関数

```
wcsftime()
```

バイト文字・ワイド文字変換関数

```
btowc() wctob()
```

変換状態関数

```
mbsinit()
```

再開始可能な多バイト文字・ワイド文字変換関数

```
mbrlen() mbrtowc() wcrtomb()
```

再開始可能な多バイト文字列・ワイド文字列変換関数

```
mbsrtowcs() wcsrtombs()
```

■wctype.hの関数

ワイド文字種分類関数

```
iswalnum() iswalpha() iswblank() iswcntrl() iswdigit() iswgraph()
iswlower() iswprint() iswpunct() iswspace() iswupper() iswxdigit()
```

拡張可能なワイド文字種分類関数

```
iswctype() wctype()
```

ワイド文字大文字小文字変換関数

```
towlower() towupper()
```

拡張可能なワイド文字大文字小文字変換関数

```
towctrans() wctrans()
```

次にワイド文字と多バイト文字を処理したサンプルプログラムを示します。これはVisual C++を使用しています。

多バイト文字を操作する

```
#include <stdio.h>
#include <stdlib.h>
#include <locale.h> /* for setlocale() */
#include <limits.h> /* MB_LEN_MAX      */

int main(void)
{
    wchar_t wdch = L'亜';             // ワイド文字
    wchar_t wdch_wk;
    char mbch[MB_LEN_MAX];            // 多バイト文字1文字分
    char ss[] = "ABCD漢字";           // 通常文字列
    wchar_t wdss[] = L"ABCD漢字";     // ワイド文字列
    char mbss[80];
    wchar_t wdss_wk[80];
    int i, n;

    setlocale(LC_ALL, "");            // 日本環境にする

    printf("(1) ワイド文字「亜」のコード=%04X¥n", (unsigned int)wdch);
```

```
    printf("(2)ワイド文字「亜」を多バイト文字に変換¥n");
    wctomb(mbch, wdch);
    printf("%02X %02X ¥n", (unsigned char)mbch[0],
                           (unsigned char)mbch[1]);

    printf("(3)多バイト文字「亜」をワイド文字に変換¥n");
    mbtowc(&wdch_wk, mbch, 2);
    printf("%04X¥n", (unsigned int)wdch_wk);

    wctomb(mbch, L'漢');
    n = mblen(mbch, MB_CUR_MAX);
    printf("(4)多バイト文字(漢)の長さ=%d¥n", n);
    wctomb(mbch, L'A');
    n = mblen(mbch, MB_CUR_MAX);
    printf("(5)多バイト文字(A)の長さ=%d¥n", n);

    printf("(6)通常文字列のコード(ABCD漢字)¥n");
    for (i=0; ss[i]; i++)
        printf("%02X ", (unsigned char)ss[i]);
    printf("¥n");

    printf("(7)ワイド文字列のコード(ABCD漢字)¥n");
    for (i=0; wdss[i]; i++)
        printf("%04X ", (unsigned int)wdss[i]);
    printf("¥n");

    printf("(8)ワイド文字列を多バイト文字列に変換¥n");
    n = wcstombs(mbss, wdss, 80);
    printf("変換数=%d ", n);
    for (i=0; mbss[i]; i++)
        printf("%02X ", (unsigned char)mbss[i]);
    printf("¥n");

    printf("(9)多バイト文字列をワイド文字列に変換¥n");
    n = mbstowcs(wdss_wk, mbss, 80);
    printf("変換数=%u ", n);
    for (i=0; wdss_wk[i]; i++)
        printf("%04X ", (unsigned int)wdss_wk[i]);
    printf("¥n");

    return 0;
}
```

実行結果：Visual C++で実行

```
(1)ワイド文字「亜」のコード=4E9C
(2)ワイド文字「亜」を多バイト文字に変換
88 9F
(3)多バイト文字「亜」をワイド文字に変換
4E9C
(4)多バイト文字(漢)の長さ=2
(5)多バイト文字(A)の長さ=1
(6)通常文字列のコード(ABCD漢字)
41 42 43 44 8A BF 8E 9A
(7)ワイド文字列のコード(ABCD漢字)
0041 0042 0043 0044 6F22 5B57
(8)ワイド文字列を多バイト文字列に変換
変換数=8 41 42 43 44 8A BF 8E 9A
(9)多バイト文字列をワイド文字列に変換
変換数=6 0041 0042 0043 0044 6F22 5B57
```

注：GNU CでUTF-8コードの場合、多バイト構成文字数が異なる。

ワイド文字用関数を使う

```
#include <stdio.h>
#include <locale.h>
#include <wchar.h>
#include <wctype.h>

int main(void)
{
    wchar_t wc = L'亜';
    wchar_t ws[] = L"ABCD漢字";
    wchar_t ws2[80];

    setlocale(LC_ALL, "");                  // 日本環境にする
    putwchar(wc); putwchar(L'¥n');          // 出力：亜
    wprintf(L"wc=%lc ws=%ls¥n", wc, ws);   // 出力：wc=亜 ws=ABCD漢字

    wcscpy(ws2, L"ワイド");
    wcscat(ws2, L"文字列¥n");
    fputws(ws2, stdout);                      // 出力：ワイド文字列
    wprintf(L"len=%d¥n", wcslen(ws));        // 出力：len=6

    if (iswalpha(ws[0])) wprintf(L"英字¥n"); // 出力：英字
    else                 wprintf(L"非英字¥n");
    ws[0] = towlower(ws[0]);
    wprintf(L"%ls¥n", ws);                   // 出力：aBCD漢字
    return 0;
}
```

注：wprintf()用法の簡単な説明。

(1)書式指定をL"～"で行なう

(2)基本的な出力書式は通常のprintf()と同じ。数値なら%dでよい

(3)ワイド文字出力書式は%lcで、ワイド文字列出力書式は%lsで行なう

160 可変個引数をもつ関数　その他の処理関数10

可変個引数の関数

C言語には**可変個引数**をもつ関数があります。printf()関数はその代表例です。書式と記述例を示します。

```
int printf(const char *format, ...);    ——書式。... が可変個引数を示す
printf("a=%d b=%d¥n", a, b);            ——記述例
```

この例では最初の固定引数を分析することで、後続するふたつの引数があることが分かります。このような可変個引数をもつ関数は自作することができます。

可変個引数用マクロ

可変個引数をもつ関数を記述するときは、次のマクロを使用します。これらはstdarg.hヘッダの中で定義されています。

void va_start(va_list ap, 〈最終引数〉);

可変個引数取得の前準備をするマクロ
va_list構造体オブジェクトに情報を設定

〈型〉 va_arg(va_list ap, 〈型〉);

可変個引数取得をするマクロ
va_list構造体オブジェクトを使って引数実体を取得

void va_end(va_list ap);

可変個引数取得の後始末をするマクロ
va_list構造体オブジェクトの情報をクリア

va_list構造体

可変個引数処理の情報を保持することができる型

注：C99ではva_copyマクロ(va_listのコピー)も導入されている。

va_startマクロの〈最終引数〉は、必ず記述しなければならない固定引数のうちの最後のものを書きます。たとえば次のようになります。myfunc()は自作の関数とします。

```
int printf(const char *format, ...);    ——format が〈最終引数〉
int myfunc(int mode, int ct, ...);      ——ct が〈最終引数〉
```

このように固定引数（複数あってよい）の最後のものを指定するので〈最終引数〉といいます。〈最終引数〉は「, ...」の直前の引数で、「どこから可変個引数がはじまるか」を調べる情報になります。

va_argマクロは実際に引数を取得するものです。〈型〉で指定したタイプの引数値を取り出します。先にva_startの設定した〈最終引数〉の次の位置から可変個引数取り出しを開始します。最後にva_endが後始末をします。

可変個引数をもつ関数

可変個引数をもつ関数はその記述例を見ると簡単に理解できます。次にふたつの固定引数と、可変個引数をもつ関数を示します。

maxmin()関数はct個のデータに対して、引数modeが'M'なら最大値、modeが'I'なら最小値（それ以外なら-1）を返します。

可変個引数をもつ関数の例

```
#include <stdio.h>
#include <stdarg.h>
int maxmin(int mode, int ct, ...);

int main(void)
{
    int n;
    n = maxmin('M', 3, 200, 300, 100);
    printf("3つの値の最大値=%d¥n", n);
    n = maxmin('I', 5, 40, 20, 50, 10, 30);
    printf("5つの値の最小値=%d¥n", n);
    return 0;
}

int maxmin(int mode, int ct, ...)    // 可変個引数を用いる関数
{
    int i, wk, ans = -1;
    va_list ap;                      // 構造体オブジェクトを確保

    va_start(ap, ct);                // 最終引数を指定してapを設定
    for (i=1; i<=ct; i++) {
        wk = va_arg(ap, int);        // int型の引数を取り出し
        if (i == 1) ans = wk;
        if (mode=='M' && ans<wk) ans = wk;           // 最大値：Max
        else if (mode=='I' && ans>wk) ans = wk;      // 最小値：mIni
    }
    va_end(ap);                      // 後処理
    return ans;
}
```

実行結果

```
3つの値の最大値=300
5つの値の最小値=10
```

データ型が混在した可変個引数をもつ関数

次のプログラムは型が混在した可変個引数をもつ関数の記述例を示しています。

udisp()関数は引数fmt内に記述された文字数で可変引数の数を判断します。また文字によって次のように引数の型を判定します。

i → **int**型の値である
d → **double**型の値である
s → **char***型の値である

データ型が混在した可変個引数をもつ関数の例

```
#include <stdio.h>
#include <stdarg.h>
void udisp(char *fmt, ...);

int main(void)
{
    udisp("idsids", 11, 22.33, "aaaa", 44, 55.66, "bbbb");
    return 0;
}

void udisp(char *fmt, ...)
{
    va_list ap;              // 構造体オブジェクトを確保
    va_start(ap, fmt);       // 最終引数を指定してapを設定
    for ( ; *fmt; ++fmt) {
        if      (*fmt == 'i') printf("%d¥n", va_arg(ap, int));
        else if (*fmt == 'd') printf("%f¥n", va_arg(ap, double));
        else if (*fmt == 's') printf("%s¥n", va_arg(ap, char *));
        else { printf("指示が不正(%c)¥n", *fmt); exit(1); }
    }
    va_end(ap);              // 後処理
}
```

実行結果

```
11
22.330000
aaaa
44
55.660000
bbbb
```
6つの各種データを表示

161 型総称マクロ

その他の処理関数11

型総称マクロ(type-generic macro)はC++言語の関数多重定義機能に相当するもので、C99で追加されました。tgmath.hヘッダをインクルードすることで利用可能になります。C言語では、ユーザーによる関数多重定義はできないので、処理系がサポートしています。

C99において数学関数と複素数関数は、型ごとに別名関数が用意されています。たとえばsin()関数は次のようになります。

```
double sin(double x);                          ——double型用
float sinf(float x);                           ——float型用
long double sinl(long double x);               ——long double型用

double complex csin(double complex z);——複素数double型用
float complex csinf(float complex z); ——複素数float型用
long double complex csinl(long double complex z);
                                     ——複素数long double型用
```

これらの関数を名前によって使い分けるのは面倒です。型総称マクロ機能を使うと、"sin"という基準となる名前(型総称マクロ名)を使うだけで、実引数のデータ型に応じて、適切な関数を自動選択してくれるようになります。

次に例を示します。型総称マクロとして使用可能なすべての名前は「L21 tgmath.h 型総称マクロ」(p. 275)で説明しています。

変数宣言

```
float f;
double d;
long double ld;
float complex fc;
double complex dc;
long double complex ldc;
```

記述と呼び出される関数

記述	実際に呼び出される関数
sin(f)	sinf(f)
sin(d)	sin(d)
sin(ld)	sinl(ld)
sin(fc)	csinf(fc)
sin(dc)	csin(dc)
sin(ldc)	csinl(ldc)

型総称マクロによる関数の自動呼び分け例

```
#include <stdio.h>
#include <math.h>
#include <tgmath.h>                              // これが必要
int main(void)
{
    float ff = 1.0f/27.0f;
    double dd = 1.0/27.0;
    long double ld = 1.0L/27.0L;

    printf("flt=%24.20f¥n",  cbrtf(ff));      // cbrtf() を呼ぶ
    printf("dbl=%24.20f¥n",  cbrt(dd));       // cbrt() を呼ぶ
    printf("ldb=%24.20llf¥n¥n",cbrtl(ld));    // cbrtl() を呼ぶ

    printf("flt=%24.20f¥n",  cbrt(ff));    //┐ cbrt() だけを指定しているが
    printf("dbl=%24.20f¥n",  cbrt(dd));    //│ 引数の型の違いで適切な関数
    printf("ldb=%24.20llf¥n",cbrt(ld));    //┘ が自動選択される
    return 0;
}
```

実行結果：GNU Cで確認した例

```
flt=  0.33333333357203709379
dbl=  0.33333333333333337034
ldb=  0.33333333333333333329

flt=  0.33333333357203709379
dbl=  0.33333333333333337034
ldb=  0.33333333333333333329
```

注：cbrt(x); はxの立方根を返す(cube root)。1.0/27.0の立方根の正解値は0.3333…である。

第19章

標準ライブラリの要約

C Quick Reference

概要

この章では、C言語に用意されている標準ライブラリ機能の要約を示します。用意されているすべての型、マクロ、プラグマ、関数について、C99機能もふくめて紹介します。

この章は関数機能の早見表として構成されています。関数形式は完全なものを提示しますが、説明は主となる機能についての1行解説です。関数にいろいろな特殊戻り値があっても言及しません。C99で追加された項目にはマークを付与しています。

complex.hとmath.hに含まれる各関数は、基本となるdouble型対応版に加えて、float型対応版と、long double型対応版が用意されています。それについては注記してあります。

なお関数形式マクロであっても、関数としての性格の強いものについては、関数部門に分類しているものがあります(例：assert, setjmp, getc, putc)。

型名やマクロ名の中には、同じものが複数のヘッダで定義されているものもあります。それらは「複数ヘッダで定義」と記してあります。該当する名前は次のものです。

複数ヘッダで定義されている名前

名前	定義されているヘッダ						
NULL	locale.h	stddef.h	stdio.h	stdlib.h	string.h	time.h	wchar.h
size_t型		stddef.h	stdio.h	stdlib.h	string.h	time.h	wchar.h
wchar_t型		stddef.h		stdlib.h			wchar.h
WCHAR_MIN						stdint.h	wchar.h
WCHAR_MAX						stdint.h	wchar.h
tm型						time.h	wchar.h
wint_t型						wctype.h	wchar.h
WEOF						wctype.h	wchar.h

L01 assert.h 診断機能 標準ライブラリの要約1

assert.hはプログラムの診断機能をサポートします。

関数

```
void assert(スカラー型 expression);
```

診断機能を付加する。expressionで条件判定する。assert()はマクロ実装である

L02 complex.h 複素数計算 標準ライブラリの要約2

complex.hは複素数計算をサポートします。このヘッダファイルはC99で追加されました。

マクロ

complex	_Complex(C99で追加された型)に展開
_Complex_I	虚数単位の値をもつ型const float _Complexの定数式に展開
imaginary	_Imaginary(C99で追加された型)の同義語にする
_Imaginary_I	虚数型サポートがあれば、虚数単位の値をもつ型const float _Imaginaryの定数式に展開
I	_Imaginary_I定義があれば_Imaginary_Iに展開。そうでなければ_Complex_Iに展開

注：complex, imaginary, Iは簡潔表現のために用意されている

プラグマ

`#pragma STDC CX_LIMITED_RANGE 状態切替指定` (状態切替指定 → ON OFF DEFAULT)

複素数の乗算、除算、絶対値処理を通常の数学式で処理してよいかを処理系に知らせる

関数(すべてC99で追加された関数)

`double cabs(double complex z);` C99

複素数絶対値を返す

`double complex cacos(double complex z);` C99

複素数逆余弦値を返す

`double complex cacosh(double complex z);` C99

複素数逆双曲線余弦値を返す

`double carg(double complex z);` C99

zの偏角の値(argument)を返す

`double complex casin(double complex z);` C99

複素数逆正弦を値を返す

`double complex casinh(double complex z);` C99

複素数逆双曲線正弦値を返す

`double complex catan(double complex z);` C99

複素数逆正接値を返す

`double complex catanh(double complex z);` C99

複素数逆双曲線正接値を返す

`double complex ccos(double complex z);` C99

複素数余弦値を返す

`double complex ccosh(double complex z);` C99

複素数双曲線余弦値を返す

`double complex cexp(double complex z);` C99

自然対数の底eのz乗(指数は複素数)を返す

`double cimag(double complex z);` C99

虚部の値を返す

`double complex clog(double complex z);` C99

zの複素数自然対数値(eを底とする)を返す

`double complex conj(double complex z);` C99

複素共役の値を返す(zの虚部の符号を反転させて複素共役を計算)

`double complex cpow(double complex x, double complex y);` C99

複素数べき乗関数値(xのy乗)を返す

`double complex cproj(double complex z);` C99

cproj()関数群は、リーマン球面上へのzの射影の値を返す

`double creal(double complex z);` C99

zの実部の値を返す

`double complex csin(double complex z);` C99

複素数正弦値を返す

`double complex csinh(double complex z);` C99

複素数双曲線正弦値を返す

`double complex csqrt(double complex z);` C99

複素数平方根値を返す

`double complex ctan(double complex z);` C99

複素数正接値を返す

`double complex ctanh(double complex z);` C99

複素数双曲線正接値を返す

これらの関数にはfloat型対応(doubleをfloatにし名前の末尾に'f'を付加)と、long double型対応(doubleをlong doubleにし名前の末尾に'l'を付加)の関数が用意されている。関数名の対応を次に示す。

関数形式例

```
double      cabs (double      complex z);     // 基本関数
float       cabsf(float       complex z);     // float型対応
long double cabsl(long double complex z);     // long double型対応
```

関数名の対応

cabs	→ cabsf, cabsl	cimag	→ cimagf, cimagl
cacos	→ cacosf, cacosl	clog	→ clogf, clogl
cacosh	→ cacoshf, cacoshl	conj	→ conjf, conjl
carg	→ cargf, cargl	cpow	→ cpowf, cpowl
casin	→ casinf, casinl	cproj	→ cprojf, cprojl
casinh	→ casinhf, casinhl	creal	→ crealf, creall
catan	→ catanf, catanl	csin	→ csinf, csinl
catanh	→ catanhf, catanhl	csinh	→ csinhf, csinhl
ccos	→ ccosf, ccosl	csqrt	→ csqrtf, csqrtl
ccosh	→ ccoshf, ccoshl	ctan	→ ctanf, ctanl
cexp	→ cexpf, cexpl	ctanh	→ ctanhf, ctanhl

L03 ctype.h 文字操作

標準ライブラリの要約3

ctype.hは文字の分類および変換についてサポートします。

関数

```
int isalnum(int c);
```

英数字なら真を返す("C"ロケールの場合)

```
int isalpha(int c);
```

英文字なら真を返す("C"ロケールの場合)

```
int isblank(int c);
```

C99

標準ブランク文字(' 'および'¥t')なら真を返す("C"ロケールの場合)

```
int iscntrl(int c);
```

制御文字なら真を返す

```
int isdigit(int c);
```

10進数字なら真を返す

```
int isgraph(int c);
```

空白を除く表示可能文字なら真を返す

```
int islower(int c);
```

小文字なら真を返す("C"ロケールの場合)

```
int isprint(int c);
```

表示可能文字(空白も含む)なら真を返す

```
int ispunct(int c);
```

区切り文字なら真を返す("C"ロケールの場合)

```
int isspace(int c);
```

空白類文字なら真を返す("C"ロケールの場合)

```
int isupper(int c);
```

大文字なら真を返す("C"ロケールの場合)

```
int isxdigit(int c);
```

16進数字なら真を返す

```
int tolower(int c);
```

大文字を小文字に変換する

```
int toupper(int c);
```

小文字を大文字に変換する

L04 errno.h エラー処理 標準ライブラリの要約4

errno.hはエラー条件の報告機能をサポートします。

マクロ

EDOM	数学関数の実引数値の定義域エラーを意味する番号を定義。errno格納用
EILSEQ	文字の表現形式エラーを意味する番号を定義。errno格納用
ERANGE	関数戻り値の値域エラーを意味する番号を定義。errno格納用
errno	int型をもつ変更可能な左辺値に展開する

L05 fenv.h 浮動小数点環境 標準ライブラリの要約5

fenv.hは浮動小数点環境へのアクセス手段をサポートします。このヘッダファイルはC99で追加されました。

型およびマクロ

fenv_t型	浮動小数点環境全体を表す型
fexcept_t型	浮動小数点状態フラグを集合的に表す型
FE_DIVBYZERO	ゼロ除算例外
FE_INEXACT	不正確結果例外
FE_INVALID	無効演算例外
FE_OVERFLOW	オーバーフロー例外
FE_UNDERFLOW	アンダーフロー例外
FE_ALL_EXCEPT	すべての例外の論理和
FE_DOWNWARD	負の無限大方向への丸め（fegetround(), fesetround()関数で利用）
FE_TONEAREST	近い値への丸め
FE_TOWARDZERO	ゼロ方向への丸め
FE_UPWARD	正の無限大方向への丸め
FE_DFL_ENV	既定の浮動小数点環境を表す

プラグマ

#pragma STDC FENV_ACCESS 状態切替指定　　（状態切替指定 → ON OFF DEFAULT）

fenv.hで提供される浮動小数点環境へのアクセスをするかしないか（最適化に関係）を指定する

関数（すべてC99で追加）

`int feclearexcept(int excepts);` C99

exceptsで指定した例外（FE_OVERFLOWなど）に対応する浮動小数点状態フラグ（浮動小数点例外の発生で設定される）をクリアする。exceptsでは「FE_OVERFLOW | FE_INEXACT」のように論理和で複数項目を指定できる（以降同じ）

`int fegetenv(fenv_t *envp);` C99

浮動小数点環境を、*envpに格納する

`int fegetexceptflag(fexcept_t *flagp, int excepts);` C99

exceptsで指定した例外に対応する状態フラグを*flagpに格納する

`int fegetround(void);` C99

丸め方向を示す値を返す

`int feholdexcept(fenv_t *envp);` C99

浮動小数点環境を*envpに保存し、状態フラグをクリアし、浮動小数点例外が発生しても実行継続するモードを設定する

`int feraiseexcept(int excepts);` C99

exceptsで指定した例外を発生する

`int fesetenv(const fenv_t *envp);` C99

envpで示す値で、現在の浮動小数点環境を設定する

`int fesetexceptflag(const fexcept_t *flagp, int excepts);` C99

exceptsで指定した例外に対応する現在の状態フラグを、*flagpに反映させる

`int fesetround(int round);` C99

roundが表す値を、現在の丸め方向として設定する

`int fetestexcept(int excepts);` C99

exceptsで指定した例外に対応する現在の状態フラグを、対応する例外マクロ値（FE_OVERFLOWなど）で返す。excepts指定と戻り値は、複数の例外マクロの論理和表現ができる

`int feupdateenv(const fenv_t *envp);` C99

その時点で生成されている浮動小数点例外を自動記憶域に保存し、envpが指す値で現在の浮動小数点環境の設定を行ない、保存していた浮動小数点例外を発生する

L06 float.h 浮動小数点型の特性 標準ライブラリの要約6

float.hは浮動小数点型の限界や特性をサポートするものです。

マクロ

FLT_EVAL_METHOD	浮動小数点数の演算時の型表現形式(範囲と精度)を決める
FLT_ROUNDS	浮動小数点型の丸め方向を決める値
FLT_RADIX	指数表現における基数
DECIMAL_DIG	処理系がサポートする最大精度の浮動小数点型の値の表現に必要な10進数表現の桁数
FLT_MANT_DIG	float型でFLT_RADIXを基数とした仮数部の桁数
DBL_MANT_DIG	同上(double版)
LDBL_MANT_DIG	同上(long double版)
FLT_DIG	float型の精度を示す10進数表現の桁数
DBL_DIG	同上(double版)
LDBL_DIG	同上(long double版)
FLT_MIN_EXP	FLT_RADIX^{x-1}が正規化された浮動小数点数となる最小の負の整数x
DBL_MIN_EXP	同上(double版)
LDBL_MIN_EXP	同上(long double版)
FLT_MAX_EXP	FLT_RADIX^{x-1}が正規化された浮動小数点数となる表現可能な有限の最大の整数x
DBL_MAX_EXP	同上(double版)
LDBL_MAX_EXP	同上(long double版)
FLT_MIN_10_EXP	10^xが正規化された浮動小数点数となる最小の負の整数x
DBL_MIN_10_EXP	同上(double版)
LDBL_MIN_10_EXP	同上(long double版)
FLT_MAX_10_EXP	10^xが正規化された浮動小数点数となる表現可能な有限の最大の整数x
DBL_MAX_10_EXP	同上(double版)
LDBL_MAX_10_EXP	同上(long double版)
FLT_MIN	float型で最小の正規化された正の浮動小数点数
DBL_MIN	同上(double版)
LDBL_MIN	同上(long double版)
FLT_MAX	float型で表現可能な有限の最大浮動小数点数
DBL_MAX	同上(double版)
LDBL_MAX	同上(long double版)
FLT_EPSILON	float型で表現可能な1より大きい最小の値と1との差
DBL_EPSILON	同上(double版)
LDBL_EPSILON	同上(long double版)

L07 inttypes.h 整数型の書式変換 標準ライブラリの要約7

inttypes.hはstdint.hに関連する機能を設定します。このヘッダファイルはC99で追加されました。

型およびマクロ

imaxdiv_t型	quot(商)とrem(剰余)というメンバをもつ構造体型

```
PRIdN PRIdLEASTN PRIdFASTN PRIdMAX PRIdPTR
PRIiN PRIiLEASTN PRIiFASTN PRIiMAX PRIiPTR
```

符号つき整数のためのfprintf書式指定子用マクロ

```
PRIoN PRIoLEASTN PRIoFASTN PRIoMAX PRIoPTR
PRIuN PRIuLEASTN PRIuFASTN PRIuMAX PRIuPTR
PRIxN PRIxLEASTN PRIxFASTN PRIxMAX PRIxPTR
PRIXN PRIXLEASTN PRIXFASTN PRIXMAX PRIXPTR
```

符号なし整数のためのfprintf書式指定子用マクロ

```
SCNdN SCNdLEASTN SCNdFASTN SCNdMAX SCNdPTR
SCNiN SCNiLEASTN SCNiFASTN SCNiMAX SCNiPTR
```

符号つき整数のためのfscanf書式指定子用マクロ

```
SCNoN SCNoLEASTN SCNoFASTN SCNoMAX SCNoPTR
SCNuN SCNuLEASTN SCNuFASTN SCNuMAX SCNuPTR
SCNxN SCNxLEASTN SCNxFASTN SCNxMAX SCNxPTR
```

符号なし整数のためのfscanf書式指定子用マクロ

注：これらのマクロはprintf()類、scanf()類関数で使用するための「長さ修飾子+変換指定子」からなる書式指定文字列に展開される。d, i, o, u, x, Xは変換指定子に対応。LEAST(最小), FAST(最速), MAX(最大), PTR(ポインタ)は<stdint.h>で定義する整数型に対応する。*N*には整数型の幅を当てる。たとえばint_fast32_t整数型の値表示にはPRIdFAST32を用いることができる。

記述例

```
intmax_t dt = 4000000000000000000LL;
printf("dt=%"PRIiMAX"¥n", dt);   // 出力：dt=4000000000000000000
```

関数（すべてC99で追加）

```
intmax_t imaxabs(intmax_t j);   C99
```

整数jの絶対値を返す

```
imaxdiv_t imaxdiv(intmax_t numer, intmax_t denom);   C99
```

numer ÷ denomの計算で商と剰余をimaxdiv_t型構造体に入れて返す

```
intmax_t strtoimax(const char *restrict nptr, char **restrict endptr, int base);   C99
uintmax_t strtoumax(const char *restrict nptr, char **restrict endptr, int base);   C99
```

文字列の最初の部分をintmax_t型、uintmax_t型の整数値に変換する。基本動作はstrtol()系関数(p. 273)と同じ

```
intmax_t wcstoimax(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int
base);   C99
uintmax_t wcstoumax(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int
base);   C99
```

ワイド文字列の最初の部分をintmax_t型、uintmax_t型の整数値に変換する。基本動作はwcstol()系関数(p. 280)と同じ

L08 iso646.h 代替つづり 標準ライブラリの要約8

iso646.hは演算子に同義の識別子を与えるものです。このヘッダファイルはC89(追補1)で追加されました。

マクロ

and	&& に展開する
and_eq	&= に展開する
bitand	& に展開する
bitor	| に展開する
compl	~ に展開する
not	! に展開する
not_eq	!= に展開する
or	|| に展開する
or_eq	|= に展開する
xor	^ に展開する
xor_eq	^= に展開する

L09 limits.h 整数型の大きさ 標準ライブラリの要約9

limits.hは整数型のサイズを定義するものです。

マクロ

CHAR_BIT	char型のビット幅
SCHAR_MIN	signed charの最小値
SCHAR_MAX	signed charの最大値
UCHAR_MAX	unsigned charの最大値
CHAR_MIN	charの最小値
CHAR_MAX	charの最大値
MB_LEN_MAX	多バイト文字の最大バイト数
SHRT_MIN	short intの最小値
SHRT_MAX	short intの最大値
USHRT_MAX	unsigned short intの最大値
INT_MIN	intの最小値
INT_MAX	intの最大値
UINT_MAX	unsigned intの最大値
LONG_MIN	long intの最小値
LONG_MAX	long intの最大値
ULONG_MAX	unsigned long intの最大値
LLONG_MIN	long long intの最小値
LLONG_MAX	long long intの最大値
ULLONG_MAX	unsigned long long intの最大値

L10 locale.h 文化圏固有操作 標準ライブラリの要約10

locale.hはロケール（文化圏固有の地域情報の反映）の操作をサポートします。

型およびマクロ

lconv型	ロケール情報を表現する型
NULL	空ポインタ定数（複数ヘッダで定義）
LC_ALL	ロケール全体を指示する定数
LC_COLLATE	文字列比較機能を指示する定数
LC_CTYPE	文字操作機能を指示する定数
LC_MONETARY	金額書式化機能を指示する定数
LC_NUMERIC	非金額数字の書式化機能を指示する定数
LC_TIME	日付・時刻表示機能を指示する定数

注：LC_ALL～LC_TIMEはsetlocale()関数で使用。

関数

```
struct lconv *localeconv(void);
```

現在のロケール情報をlconv構造体に設定し、それへのポインタを返す

```
char *setlocale(int category, const char *locale);
```

categoryで指定するカテゴリを、localeで指定したロケール（地域仕様）に設定する

L11 math.h 数学関数

標準ライブラリの要約11

math.hは数学関数とその関連機能をサポートします。

型およびマクロ

float_t型	float型以上の幅をもち、処理系においてもっとも効率的に処理できる型
double_t型	double型以上の幅をもち、処理系においてもっとも効率的に処理できる型
HUGE_VAL	double型で表現可能な最大値(float型として表現できるとは限らない正のdouble型の式に展開)
HUGE_VALF	float型対応のHUGE_VAL C99
HUGE_VALL	long double型対応のHUGE_VAL C99
INFINITY	無限大を意味する浮動小数点定数
NAN	非数(Not-a-Number)を意味する表現。qNaNに展開
FP_INFINITE	浮動小数点数が無限大であることを示す整数定数
FP_NAN	浮動小数点数がNaNであることを示す整数定数
FP_NORMAL	浮動小数点数が正規化数であることを示す整数定数
FP_SUBNORMAL	浮動小数点数が非正規化数であることを示す整数定数
FP_ZERO	浮動小数点数が0であることを示す整数定数
FP_FAST_FMA	fma()関数によるdouble型演算がハードウェア命令を利用して高速実行されることを表す
FP_FAST_FMAF	同上(float版)
FP_FAST_FMAL	同上(long double版)
FP_ILOGB0	ilogb(0)の戻り値となる整数定数式に展開
FP_ILOGBNAN	ilogb(NaN)の戻り値となる整数定数式に展開
MATH_ERRNO	定数1に展開。math_errhandlingのマスク値として使用
MATH_ERREXCEPT	定数2に展開。math_errhandlingのマスク値として使用
math_errhandling	数学エラー種別表現に使用。MATH_ERRNOとMATH_ERREXCEPTを表現
fpclassify(x)	値xをカテゴリに分類して返す(FP_NAN, FP_ZEROなど)
isfinite(x)	値xが有限の値(無限大やNaNなどでない)かどうかを判定する
isinf(x)	値xが無限大であるかどうかを判定する
isnan(x)	値xがNaNであるかどうかを判定する
isnormal(x)	値xが正規化数であるかどうかを判定する
signbit(x)	値xの符号が負かどうかを判定する(負のとき非0を返す)
isgreater(x,y)	(x)>(y)かどうかを判定する
isgreaterequal(x,y)	(x)>=(y)かどうかを判定する
isless(x,y)	(x)<(y)かどうかを判定する
islessequal(x,y)	(x)<=(y)かどうかを判定する
islessgreater(x,y)	(x)<(y)||(x)>(y)かどうかを判定する
isunordered(x,y)	xとyに順序がない場合(片方がNaNなど) 1を返す

注1:NaNは非数(Not-a-Number)を意味する表現形式。qNaN(quiet NaN)は例外を起こさないNaN, sNaN(signaling NaN)は例外を起こすNaN。

注2:引数x, yは浮動小数点型である。

注3:5つの比較マクロは引数が不適切(NaNなど)であるときも無効演算浮動小数点例外を発生しない。

プラグマ

```
#pragma STDC FP_CONTRACT 状態切替指定          (状態切替指定 → ON OFF DEFAULT)
```

式を短縮(高速化につながる)することを認めるかどうかを指定する

関数

```
double acos(double x);
```

逆余弦値を返す

```
double acosh(double x);
```
C99

双曲線逆余弦値を返す

```
double asin(double x);
```

逆正弦値を返す

```
double asinh(double x);
```
C99

双曲線逆正弦値を返す

```
double atan(double x);
```

逆正接値を返す

```
double atan2(double y, double x);
```

y/xの逆正接値を返す

```
double atanh(double x);
```
C99

双曲線逆正接値を返す

```
double cbrt(double x);
```
C99

xの立方根を返す

```
double ceil(double x);
```

x以上の最小の整数値を返す

```
double copysign(double x, double y);
```
C99

xの絶対値にyの符号をつけた値を返す

```
double cos(double x);
```

余弦値を返す

```
double cosh(double x);
```

双曲線余弦値を返す

```
double erf(double x);
```
C99

xの誤差関数値を返す

```
double erfc(double x);
```
C99

xの余誤差関数値を返す

```
double exp(double x);
```

自然対数の底eのx乗を返す

```
double exp2(double x);
```
C99

2のx乗を返す

```
double expm1(double x);
```
C99

自然対数の底eのx乗から1を引いた値を返す

```
double fabs(double x);
```

xの絶対値を返す

```
double fdim(double x, double y);
```
C99

2個の引数の正の差を返す。x>yならx-yを、x<=yなら+0を返す

```
double floor(double x);
```

x以下の最大の整数値を返す

```
double fma(double x, double y, double z);
```
C99

(x×y)+zを計算する。途中精度を落とさない計算を行なう

```
double fmax(double x, double y);
```
C99

大きい方の値を返す

```
double fmin(double x, double y);
```
C99

小さい方の値を返す

```
double fmod(double x, double y);
```

x/yの浮動小数点剰余を返す。結果はxと同じ符号をもち、yの絶対値より小さい絶対値をもつ

```
double frexp(double value, int *exp);
```

value = x * 2^nという計算を行ないxを返す。*expにnを入れる

```
double hypot(double x, double y);
```
C99

xの2乗とyの2乗の和、の平方根を計算する。計算途中でのオーバーフロー、アンダーフローを抑止する

```
int ilogb(double x);
```
C99

xの符号つき指数をint値として返す。logb()関数を呼びintにキャストすることと等価

```
double ldexp(double x, int exp);
```

x × 2^{exp}という値を返す

```
double lgamma(double x);
```
C99

xのガンマ関数の絶対値の自然対数値を返す

```
long long int llrint(double x);
```
C99

現在の丸め方向設定にしたがって、xをlong long int型整数値に丸める

```
long long int llround(double x);
```
C99

xをもっとも近いlong long int型整数値に丸める。値が中間にある場合は0から遠い方向の値を選ぶ。

```
double log(double x);
```

xの自然対数値(eを底とする)を返す

```
double log10(double x);
```

常用対数値(10を底とする)を返す

```
double log1p(double x);
```
C99

(x+1)という値の自然対数値(eを底とする)を返す

```
double log2(double x);
```
C99

xの対数値(2を底とする)を返す

```
double logb(double x);
```
C99

xの符号つき指数を返す

```
long int lrint(double x);
```
C99

現在の丸め方向設定にしたがって、xをlong int型整数値に丸める

```
long int lround(double x);
```
C99

xをもっとも近いlong int型整数値に丸める。値が中間にある場合は0から遠い方向の値を選ぶ

```
double modf(double value, double *iptr);
```

valueの符号つき小数部を返す。*iptrに整数部を格納する

```
double nan(const char *tagp);
```
C99

tagpが示す内容をもつqNaNを返す。qNaNが非サポートのときは0を返す。nan("n文字列")はstrtod("NAN(n文字列)", (char**)NULL)と等価

```
double nearbyint(double x);
```
C99

現在の丸め方向設定にしたがって、xを浮動小数点形式の整数値に丸める

```
double nextafter(double x, double y);
```
C99

xを、y方向に、最小の値変化をさせた値を返す。x=yのときはyを返す

```
double nexttoward(double x, long double y);
```
C99

2番目の引数がlong double型であり、x=yのときはyをその関数型にキャストして返すこと以外は、nextafter()関数群と等価

```
double pow(double x, double y);
```

xのy乗の値を返す

```
double remainder(double x, double y);
```
C99

剰余値x REM yを返す。x÷yにもっとも近い整数値をn(値が中間にある場合は偶数を選ぶ)としたとき、戻り値はx-n×yになる

```
double remquo(double x, double y, int *quo);
```
C99

remainderと同じ剰余値を返す。さらに符号がx/yと同じでx/yの商(整数)の絶対値と、2のn乗(nは3以上の処理系定義整数値)を法として合同な整数値(意味的にはx/yの商の下位ビット)を*quoに格納する

```
double rint(double x);
```
C99

nearbyint()と同じ整数値丸め計算をする。結果と実引数値が相違したとき「不正確結果」浮動小数点例外を生成することが異なる

```
double round(double x);
```
C99

xをもっとも近い整数値に丸める。値が中間(xx.5という値)にある場合は0から遠い方向の値にする(例:2.5は3.0にする)

```
double scalbln(double x, long int n);
```
C99

scalbn()関数と同じ処理。第2引数がlong

```
double scalbn(double x, int n);
```
C99

x × FLT_RADIXのn乗を返す。この関数は効率よい計算を意図している。第2引数がint

```
double sin(double x);
```

正弦値を返す

```
double sinh(double x);
```

双曲線正弦値を返す

```
double sqrt(double x);
```

xの平方根を返す

```
double tan(double x);
```

正接値を返す

```
double tanh(double x);
```

双曲線正接値を返す

```
double tgamma(double x);
```
C99

xのガンマ関数値を返す

```
double trunc(double x);
```
C99

xをゼロ方向のもっとも近い整数値に丸める

これらの関数にはfloat型対応(doubleをfloatにし名前の末尾に'f'を付加)と、long double型対応(doubleをlong doubleにし名前の末尾に'l'を付加)の関数が用意されている(C99で追加)。関数名の対応を次に示す。

関数形式例

```
double      acos (double      x);   // 基本関数
float       acosf(float       x);   // float型対応
long double acosl(long double x);   // long double型対応
```

関数名の対応

acos	→ acosf, acosl
acosh	→ acoshf, acoshl
asin	→ asinf, asinl
asinh	→ asinhf, asinhl
atan	→ atanf, atanl
atan2	→ atan2f, atan2l
atanh	→ atanhf, atanhl
cbrt	→ cbrtf, cbrtl
ceil	→ ceilf, ceill
copysign	→ copysignf, copysignl
cos	→ cosf, cosl
cosh	→ coshf, coshl
erf	→ erff, erfl
erfc	→ erfcf, erfcl
exp	→ expf, expl
exp2	→ exp2f, exp2l
expm1	→ expm1f, expm1l
fabs	→ fabsf, fabsl
fdim	→ fdimf, fdiml
floor	→ floorf, floorl
fma	→ fmaf, fmal
fmax	→ fmaxf, fmaxl
fmin	→ fminf, fminl
fmod	→ fmodf, fmodl
frexp	→ frexpf, frexpl
hypot	→ hypotf, hypotl
ilogb	→ ilogbf, ilogbl
ldexp	→ ldexpf, ldexpl
lgamma	→ lgammaf, lgammal
llrint	→ llrintf, llrintl
llround	→ llroundf, llroundl
log	→ logf, logl

log10	→ log10f, log10l
log1p	→ log1pf, log1pl
log2	→ log2f, log2l
logb	→ logbf, logbl
lrint	→ lrintf, lrintl
lround	→ lroundf, lroundl
modf	→ modff, modfl
nan	→ nanf, nanl
nearbyint	→ nearbyintf, nearbyintl
nextafter	→ nextafterf, nextafterl
nexttoward	→ nexttowardf, nexttowardl(注)
pow	→ powf, powl
remainder	→ remainderf, remainderl
remquo	→ remquof, remquol
rint	→ rintf, rintl
round	→ roundf, roundl
scalbln	→ scalblnf, scalblnl
scalbn	→ scalbnf, scalbnl
sin	→ sinf, sinl
sinh	→ sinhf, sinhl
sqrt	→ sqrtf, sqrtl
tan	→ tanf, tanl
tanh	→ tanhf, tanhl
tgamma	→ tgammaf, tgammal
trunc	→ truncf, truncl

注：第2引数はlong double型固定。

L12 setjmp.h 非局所分岐 標準ライブラリの要約12

setjmp.hは非局所的ジャンプをサポートします。

型

jmp_buf型	非局所分岐の呼び出し環境を復元するために必要な情報を保持することができる型

関数

```
void longjmp(jmp_buf env, int val);
```

setjmp()で保存されたenvで示す位置にジャンプする。valを対応するsetjmp()の戻り値にする

```
int setjmp(jmp_buf env);
```

呼び出し環境をenvに保存する。setjmp()はマクロ実装である

L13 signal.h シグナル操作 標準ライブラリの要約13

signal.hは実行中に生じるシグナル情報の処理機能をサポートします。

型およびマクロ

sig_atomic_t型	非同期割込みがあってもひとつの不可分な実体としてアクセスできる整数型
SIGABRT	異常終了を示す整数定数
SIGFPE	算術エラーを示す整数定数
SIGILL	不正命令などを示す整数定数
SIGINT	割込みを示す整数定数
SIGSEGV	メモリへの不正アクセスを示す整数定数
SIGTERM	終了要求を示す整数定数
SIG_DFL	signal()関数で「標準の処理」に戻すことを指示する引数に利用
SIG_ERR	signal()関数でエラーが発生した場合の戻り値
SIG_IGN	signal()関数でシグナル無視を指示する引数に利用する

関数

```
void (*signal(int sig, void (*func)(int)))(int);
```

シグナルsigを受け取ったときに実行させたい処理を設定する

```
int raise(int sig);
```

sigで指定したシグナル動作を実行する

L14 stdarg.h 可変個数の実引数操作 標準ライブラリの要約14

stdarg.hは関数の可変個引数処理をサポートします。

マクロ

void va_start(va_list ap, 最終仮引数)	可変個引数処理の準備
型 va_arg(va_list ap, 型)	「次の引数」を取得
void va_end(va_list ap)	可変個引数処理の後始末
void va_copy(va_list dest, va_list src)	情報srcをdestに複製する C99

L15 stdbool.h 論理型および論理値 標準ライブラリの要約15

stdbool.hは論理型関連機能をサポートします。このヘッダファイルはC99で追加されました。

マクロ

bool	_Boolに展開
true	定数1に展開
false	定数0に展開
__bool_true_false_are_defined	定数1に展開

L16 stddef.h 共通の定義 標準ライブラリの要約16

stddef.hは共通利用目的の型とマクロを定義するものです。

型およびマクロ

ptrdiff_t型	ふたつのポインタの差を表現できる符号つき整数型
size_t型	sizeof演算子の結果を表現できる符号なし整数型(複数ヘッダで定義)
wchar_t型	ワイド文字を表現できる整数型(複数ヘッダで定義)
NULL	空ポインタ定数(複数ヘッダで定義)
offsetof(型, メンバ)	「型」で示す構造体のメンバの先頭からのオフセット番地を返す

L17 stdint.h 整数型処理 標準ライブラリの要約17

stdint.hはいろいろな拡張整数型をサポートします。このヘッダファイルはC99で追加されました。

処理系の基本データ型の実装は同一でない(例:int型の幅)ため、可搬性に問題があります。このヘッダで定義する型は、可搬性を高める目的をもちます。

たとえば16ビットデータを操作するとき、「16ビット整数データを最小の記憶域サイズで処理するにはshort int型がよい」、また「16ビット整数データを最速で処理するにはint型がよい」、などの判断がでてきます。

このヘッダで定義する型は、それらの用途に適した実データ型を適切に選択してくれる役目をもちます。

定義される型には次の分類があります。例はint幅が32ビットである処理系での定義例です。

厳密な幅をもつ整数型定義

```
例：typedef short int               int16_t;
例：typedef unsigned short int uint16_t;
```

最小幅をもつ整数型

```
例：typedef short int               int_least16_t;
例：typedef unsigned short int uint_least16_t;
```

最速処理をする整数型

```
例：typedef int                     int_fast16_t;    // 高速処理のためint型
例：typedef unsigned int            uint_fast16_t;
```

最大の幅をもつ整数型

```
例：typedef long long int           intmax_t;
例：typedef unsigned long long int uintmax_t;
```

任意のポインタを格納可能な型

```
例：typedef int                     intptr_t;
例：typedef unsigned int            uintptr_t;
```

型（注：*N*はビット数で置換する）

int*N*_t型 uint*N*_t型	厳密に*N*ビットの幅をもつ符号つき整数型。例：int8_t 厳密に*N*ビットの幅をもつ符号なし整数型。例：uint8_t 処理系が提供する整数型に合わせて、対応する型定義名を定義しなければならない
int_least*N*_t型 uint_least*N*_t型	少なくとも*N*ビットの幅をもつ符号つき整数型 少なくとも*N*ビットの幅をもつ符号なし整数型 このタイプとして次の型が必須である int_least8_t uint_least8_t int_least16_t uint_least16_t int_least32_t uint_least32_t int_least64_t uint_least64_t
int_fast*N*_t型 uint_fast*N*_t型	少なくとも*N*ビットの幅をもつ最速処理できる符号つき整数型 少なくとも*N*ビットの幅をもつ最速処理できる符号なし整数型 このタイプとして次の型が必須である int_fast8_t uint_fast8_t int_fast16_t uint_fast16_t int_fast32_t uint_fast32_t int_fast64_t uint_fast64_t
intmax_t型 uintmax_t型	すべての符号つき整数型のすべての値を表現可能な符号つき整数型 すべての符号なし整数型のすべての値を表現可能な符号なし整数型 これらの型は必須である
intptr_t型 uintptr_t型	ポインタを正確に保持できる符号つき整数型 ポインタを正確に保持できる符号なし整数型 これらの型は省略可能である

マクロ（注：*N*はビット数で置換する）

INT*N*_MIN	幅指定符号つき整数型の最小値
INT*N*_MAX	幅指定符号つき整数型の最大値
UINT*N*_MAX	幅指定符号なし整数型の最大値
INT_LEAST*N*_MIN	最小幅指定符号つき整数型の最小値
INT_LEAST*N*_MAX	最小幅指定符号つき整数型の最大値
UINT_LEAST*N*_MAX	最小幅指定符号なし整数型の最大値

INT_FAST*N*_MIN	最速最小幅指定符号つき整数型の最小値
INT_FAST*N*_MAX	最速最小幅指定符号つき整数型の最大値
UINT_FAST*N*_MAX	最速最小幅指定符号なし整数型の最大値
INTPTR_MIN	ポインタ保持可能な符号つき整数型の最小値
INTPTR_MAX	ポインタ保持可能な符号つき整数型の最大値
UINTPTR_MAX	ポインタ保持可能な符号なし整数型の最大値
INTMAX_MIN	最大幅符号つき整数型の最小値
INTMAX_MAX	最大幅符号つき整数型の最大値
UINTMAX_MAX	最大幅符号なし整数型の最大値
PTRDIFF_MIN	ptrdiff_t型の最小値
PTRDIFF_MAX	ptrdiff_t型の最大値
SIG_ATOMIC_MIN	sig_atomic_t型の最小値
SIG_ATOMIC_MAX	sig_atomic_t型の最大値
SIZE_MAX	size_t型の最大値
WCHAR_MIN	wchar_t型の最小値（複数ヘッダで定義）
WCHAR_MAX	wchar_t型の最大値（複数ヘッダで定義）
WINT_MIN	wint_t型の最小値
WINT_MAX	wint_t型の最大値
INT*N*_C(値)	同じ値をもつint_least*N*_t型の整数定数式に展開
UINT*N*_C(値)	同じ値をもつuint_least*N*_t型の整数定数式に展開
INTMAX_C(値)	同じ値をもつintmax_t型の整数定数式に展開
UINTMAX_C(値)	同じ値をもつuintmax_t型の整数定数式に展開

L18 stdio.h 入出力処理　標準ライブラリの要約18

stdio.hは入出力関連の機能をサポートします。

型およびマクロ

size_t型	sizeof演算子の結果を表現できる符号なし整数型（複数ヘッダで定義）
FILE型	ストリーム制御用の情報を記録できる型
fpos_t型	ファイルの中の任意の位置情報を記録できる型。fgetpos(), fsetpos()で使用
NULL	空ポインタ定数（複数ヘッダで定義）
_IOFBF	完全バッファリングを示す定数（setvbuf()関数で使用）
_IOLBF	行バッファリングを示す定数（setvbuf()関数で使用）
_IONBF	非バッファリングを示す定数（setvbuf()関数で使用）
BUFSIZ	バッファサイズを示す定数（setbuf()関数で使用）
EOF	ファイルの終わり（end-of-file）を示す定数
FOPEN_MAX	同時にオープン可能であることを保証するファイル数
FILENAME_MAX	ファイル名の使用可能最大長を示す定数
L_tmpnam	tmpnam()関数が生成するファイル名を保持するに必要なchar型の配列長を示す定数
SEEK_CUR	移動基点〈現在位置から〉を示す定数（fseek()関数で使用）
SEEK_END	移動基点〈終端位置から〉を示す定数（fseek()関数で使用）
SEEK_SET	移動基点〈先頭位置から〉を示す定数（fseek()関数で使用）
TMP_MAX	tmpnam()関数が生成するファイル名の最大個数を表す定数
stderr	標準エラー出力ストリームへのポインタ
stdin	標準入力ストリームへのポインタ
stdout	標準出力ストリームへのポインタ

関数

```
void clearerr(FILE *stream);
```

指定ストリームのファイル終了表示子およびエラー表示子をクリアする

```
int fclose(FILE *stream);
```

指定ストリームを終了する(ファイルをクローズする)

```
int feof(FILE *stream);
```

指定ストリームのファイル終了表示子がセットされていれば非0を返す

```
int ferror(FILE *stream);
```

指定ストリームのエラー表示子がセットされていれば非0を返す

```
int fflush(FILE *stream);
```

指定ストリーム(出力系動作のとき)の未書き込みのデータをファイルに書き込む

```
int fgetc(FILE *stream);
```

指定ストリームから1文字取得して返す

```
int fgetpos(FILE * restrict stream, fpos_t * restrict pos);
```

指定ストリームのファイル位置表示子情報を*posに格納する

```
char *fgets(char * restrict s, int n, FILE * restrict stream);
```

指定ストリームから文字列を読み込んで配列sに格納する

```
FILE *fopen(const char * restrict filename, const char * restrict mode);
```

filenameで示すファイルをモードmodeでオープンする

```
int fprintf(FILE * restrict stream, const char * restrict format, ...);
```

formatで示す書式で指定ストリームに出力する

```
int fputc(int c, FILE *stream);
```

指定ストリームに文字cを書き込む

```
int fputs(const char * restrict s, FILE * restrict stream);
```

指定ストリームに文字列sを書き込む

```
size_t fread(void * restrict ptr, size_t size, size_t nmemb, FILE * restrict
stream);
```

指定ストリームから大きさsizeの要素を最大nmemb個読み取り、ptrが指す配列に入れる

```
FILE *freopen(const char * restrict filename, const char * restrict mode, FILE *
restrict stream);
```

指定ストリームをfilenameとmodeで再オープンする

```
int fscanf(FILE * restrict stream, const char * restrict format, ...);
```

formatで示す書式で指定ストリームから入力する

```
int fseek(FILE *stream, long int offset, int whence);
```

指定ストリームのファイル位置表示子の値を変更する

```
int fsetpos(FILE *stream, const fpos_t *pos);
```

*posの値で指定ストリームのファイル位置表示子を設定する

```
long int ftell(FILE *stream);
```

指定ストリームのファイル位置表示子の値を返す

```
size_t fwrite(const void * restrict ptr, size_t size, size_t nmemb, FILE *
restrict stream);
```

指定ストリームに、ptrが指す配列から、大きさsizeの要素を最大nmemb個書き込む

```
int getc(FILE *stream);
```

fgetc()と等価とする。マクロ実装されることがある

```
int getchar(void);
```

標準入力(stdin)から1文字取得して返す

```
char *gets(char *s);
```

標準入力(stdin)から文字列を読み込んで配列sに格納する

```
void perror(const char *s);
```

標準エラー出力(stderr)に、文字列sおよび現在のerrnoに対応するエラーメッセージを出力する

```
int printf(const char * restrict format, ...);
```

formatで示す書式で標準出力(stdout)に出力する

```
int putc(int c, FILE *stream);
```

fputc()と等価である。マクロ実装されることがある

```
int putchar(int c);
```

標準出力(stdout)に1文字出力する

```
int puts(const char *s);
```

標準出力(stdout)に文字列sを書き込む。出力の最後に改行文字を追加する

```
int remove(const char *filename);
```

filenameで示すファイルを削除する

```
int rename(const char *old, const char *new);
```

oldで示すファイル名を、newで示すファイル名に変更する

```
void rewind(FILE *stream);
```

指定ストリームのファイル位置表示子を先頭に位置付ける。エラー指示子もクリアする

```
int scanf(const char * restrict format, ...);
```

formatで示す書式で標準入力(stdin)から入力する

```
void setbuf(FILE * restrict stream, char * restrict buf);
```

modeを値_IOFBF, sizeを値BUFSIZとしたsetvbuf()と同じである(値は返さない)

```
int setvbuf(FILE * restrict stream, char * restrict buf, int mode, size_t size);
```

指定ストリームの入出力バッファをbufに変更する。bufのサイズをsizeで知らせる。modeでバッファ処理方法を指定する

```
int snprintf(char * restrict s, size_t n, const char * restrict format, ...);
```
C99

sprint()関数と同じだが、書き込む最大文字数をnで指定できる

```
int sprintf(char * restrict s, const char * restrict format, ...);
```

formatで示す書式でsで指定する配列に出力する

```
int sscanf(const char * restrict s, const char * restrict format, ...);
```

formatで示す書式で、sで示す文字列から入力する

```
FILE *tmpfile(void);
```

既存のファイルとは異なる一時バイナリファイルを生成する

```
char *tmpnam(char *s);
```

既存のファイル名と一致しないファイル名文字列を生成し、それへのポインタを返す

```
int ungetc(int c, FILE *stream);
```

指定ストリームに文字cを押し戻す

```
int vfprintf(FILE * restrict stream, const char * restrict format, va_list arg);
```

可変個数の実引数並びをargで置き換えたfprintf()と等価である

```
int vfscanf(FILE * restrict stream, const char * restrict format, va_list arg);
```
C99

可変個数の実引数並びをargで置き換えたfscanf()と等価である

```
int vprintf(const char * restrict format, va_list arg);
```

可変個数の実引数並びをargで置き換えたprintf()と等価である

```
int vscanf(const char * restrict format, va_list arg);
```
C99

可変個数の実引数並びをargで置き換えたscanf()と等価である

```
int vsnprintf(char * restrict s, size_t n, const char * restrict format, va_list
arg);
```
C99

可変個数の実引数並びをargで置き換えたsnprintf()と等価である

```
int vsprintf(char * restrict s, const char * restrict format, va_list arg);
```

可変個数の実引数並びをargで置き換えたsprintf()と等価である

```
int vsscanf(const char * restrict s, const char * restrict format, va_list arg);
```
C99

可変個数の実引数並びをargで置き換えたsscanf()と等価である

L19 stdlib.h 一般ユーティリティ 標準ライブラリの要約19

stdlib.hは数値変換や記憶割り当てなどの機能をサポートします。

型およびマクロ

size_t型	sizeof演算子の結果を表現できる符号なし整数型(複数ヘッダで定義)
wchar_t型	ワイド文字を表現できる整数型(複数ヘッダで定義)
div_t型	div()関数が返す値を表現する型(メンバquotとremをもつ)
ldiv_t型	ldiv()関数が返す値を表現する型
lldiv_t型	lldiv()関数が返す値を表現する型
NULL	空ポインタ定数(複数ヘッダで定義)
EXIT_FAILURE	失敗終了状態を示す定数
EXIT_SUCCESS	成功終了状態を示す定数
RAND_MAX	rand()関数が返す最大の値
MB_CUR_MAX	現在のロケールにおける多バイト文字の最大バイト数

関数

```
void _Exit(int status);  C99
```

プログラムを正常終了する。exit()関数と異なり、atexit()関数およびsignal()関数で登録された関数を呼び出すなどの処理は行なわない

```
void abort(void);
```

プログラムを意図的に異常終了させる(シグナルSIGABRTを発生する)

```
int abs(int j);
```

整数jの絶対値を返す。→labs(), llabs()

```
int atexit(void (*func)(void));
```

プログラムの正常終了時に呼び出す関数を登録する

```
double atof(const char *nptr);
```

nptrで示す文字列の最初の部分をdouble型の値に変換する

```
int atoi(const char *nptr);
long int atol(const char *nptr);
long long int atoll(const char *nptr);  C99
```

nptrで示す文字列の最初の部分を整数値に変換する

```
void *bsearch(const void *key, const void *base, size_t nmemb, size_t size, int
(*compar)(const void *, const void *));
```

先頭位置baseからnmemb個のオブジェクト(1要素の大きさはsize)の配列からkeyで示すデータを探索する。comparで比較用関数を指定する

```
void *calloc(size_t nmemb, size_t size);
```

大きさがsizeであるオブジェクトnmemb個分の配列領域を割り付け、0で初期化する

```
div_t div(int numer, int denom);
```

numer÷denomを計算し、商(quot)と剰余(rem)を格納した構造体を返す。→ldiv(), lldiv()

```
void exit(int status);
```

プログラムを正常終了する。このときstatus値をホスト環境に戻す。atexit()関数およびsignal()関数で登録された関数を呼び出し、開かれているストリームを正常終了させるなどの後始末サービスを行なう

```
void free(void *ptr);
```

ptrが指すメモリ領域を解放する

```
char *getenv(const char *name);
```

nameで示す環境変数(ホスト環境が管理)を探し、その定義文字列を指すポインタを返す

```
long int labs(long int j);
long long int llabs(long long int j);  C99
```

整数jの絶対値を返す。→abs()

```
ldiv_t ldiv(long int numer, long int denom);
lldiv_t lldiv(long long int numer, long long int denom);  C99
```

numer÷denomを計算し、商(quot)と剰余(rem)を格納した構造体を返す。→div()

```
void *malloc(size_t size);
```

大きさがsizeであるメモリ領域を割り付ける

```
int mblen(const char *s, size_t n);
```

多バイト文字列sの先頭n字までを検査し、多バイト文字を構成するバイト数を返す

```
size_t mbstowcs(wchar_t * restrict pwcs, const char * restrict s, size_t n);
```

多バイト文字列sをワイド文字列に変換し、n個以下の文字をwcsに格納する。格納したワイド文字数を返す

```
int mbtowc(wchar_t * restrict pwc, const char * restrict s, size_t n);
```

sの指す文字列の最大nバイトを検査し、多バイト文字をワイド文字に変換してpwcに格納する。多バイト文字の構成バイト数を返す

```
void qsort(void *base, size_t nmemb, size_t size, int(*compar)(const void *, const
void *));
```

先頭位置baseからnmemb個のオブジェクト(1要素の大きさはsize)の配列を整列する。comparで比較用関数を指定する

```
int rand(void);
```

0以上RAND_MAX以下の範囲の擬似乱数整数を返す

```
void *realloc(void *ptr, size_t size);
```

ptrが指す古いオブジェクト内容を維持して、新しい大きさsizeでメモリを再割り付けする

```
void srand(unsigned int seed);
```

seedを種として擬似乱数系列を変更する

```
double strtod(const char * restrict nptr, char ** restrict endptr);
float strtof(const char * restrict nptr, char ** restrict endptr);  C99
long double strtold(const char * restrict nptr, char ** restrict endptr);  C99
```

nptrで示す文字列の最初の部分を浮動小数点数に変換する。未変換文字列へのポインタをendptrに入れる

```
long int strtol(const char * restrict nptr, char ** restrict endptr, int base);
long long int strtoll(const char * restrict nptr, char ** restrict endptr, int
base); C99
unsigned long int strtoul(const char * restrict nptr, char ** restrict endptr, int
base);
unsigned long long int strtoull(const char * restrict nptr, char ** restrict
endptr, int base); C99
```

nptrで示す文字列の最初の部分を整数値に変換する。未変換文字列へのポインタをendptrに入れる

```
int system(const char *string);
```

コマンドプロセッサを呼び、stringで示す文字列をコマンドとして実行する

```
size_t wcstombs(char * restrict s, const wchar_t * restrict pwcs, size_t n);
```

ワイド文字列pwcsを多バイト文字列に変換してsに格納する。sに書き込んだバイト数を返す

```
int wctomb(char *s, wchar_t wc);
```

ワイド文字wcを多バイト文字に変換してsに格納し、sに書き込んだバイト数を返す

L20 string.h 文字列操作 標準ライブラリの要約20

string.hは文字列処理機能をサポートします。

型およびマクロ

size_t型	sizeof演算子の結果を表現できる符号なし整数型（複数ヘッダで定義）
NULL	空ポインタ定数（複数ヘッダで定義）

関数

```
void *memchr(const void *s, int c, size_t n);
```

sが指すオブジェクトの先頭からn文字をサーチし文字cのある位置を返す

```
int memcmp(const void *s1, const void *s2, size_t n);
```

s1とs2が指すオブジェクトの先頭からn文字を比較する

```
void *memcpy(void * restrict s1, const void * restrict s2, size_t n);
```

s1が指すオブジェクトに、s2が指すオブジェクトのn文字をコピーする

```
void *memmove(void *s1, const void *s2, size_t n);
```

memcpy()と同じ動作。コピー元とコピー先が重なっていても正しくコピーする

```
void *memset(void *s, int c, size_t n);
```

sが指すオブジェクトの最初のn文字を文字c(unsigned char型に変換）で初期化する

```
char *strcat(char * restrict s1, const char * restrict s2);
```

文字列s1の最後に文字列s2を連結する

```
char *strchr(const char *s, int c);
```

文字列sの中で文字cが最初に現れる位置を返す

```
int strcmp(const char *s1, const char *s2);
```

文字列s1と文字列s2を比較する

```
int strcoll(const char *s1, const char *s2);
```

文字列s1と文字列s2の内容を現在の地域仕様にしたがって比較する

```
char *strcpy(char * restrict s1, const char * restrict s2);
```

配列s1に文字列s2をコピーする

```
size_t strcspn(const char *s1, const char *s2);
```

文字列s1の中で、文字列s2にふくまれない文字だけで構成される先頭部分の文字数を返す

```
char *strerror(int errnum);
```

エラー番号errnumに対応するメッセージ文字列へのポインタを返す

```
size_t strlen(const char *s);
```

文字列sの長さを計算して返す

```
char *strncat(char * restrict s1, const char * restrict s2, size_t n);
```

文字列s1の最後に文字列s2の先頭n文字を連結する

```
int strncmp(const char *s1, const char *s2, size_t n);
```

文字列s1と文字列s2の先頭n文字を比較する

```
char *strncpy(char * restrict s1, const char * restrict s2, size_t n);
```

配列s1に文字列s2の先頭n文字をコピーする

```
char *strpbrk(const char *s1, const char *s2);
```

文字列s1の中で、文字列s2内のどれかの文字が最初に現れる位置を返す

```
char *strrchr(const char *s, int c);
```

文字列sの中で文字cが最後に現れる位置を返す

```
size_t strspn(const char *s1, const char *s2);
```

文字列s1の中で、文字列s2にふくまれる文字だけで構成される先頭部分の文字数を返す

```
char *strstr(const char *s1, const char *s2);
```

文字列s1の中で文字列s2が最初に現れる位置を返す

```
char *strtok(char * restrict s1, const char * restrict s2);
```

文字列s2に含まれる文字を分離記号として、文字列s1を分離して返す

```
size_t strxfrm(char * restrict s1, const char * restrict s2, size_t n);
```

文字列s2を地域仕様にもとづいて変換し、変換結果の文字列の先頭n文字までを配列s1に格納する

L21 tgmath.h 型総称マクロ 標準ライブラリの要約21

tgmath.h型総称マクロ機能をサポートします。このヘッダファイルはC99で追加されました。「161 型総称マクロ」(p. 244)に関連説明があります。型総称マクロ名とそれに対応する関数名の一覧を次に示します。

型総称マクロ1

マクロ名に対応する関数がmath.hとcomplex.hにあるもの。

注：modf()関数は型総称マクロをもたない。

型総称マクロ	math.h対応関数			complex.h対応関数		
acos	acos	acosf	acosl	cacos	cacosf	cacosl
asin	asin	asinf	asinl	casin	casinf	casinl
atan	atan	atanf	atanl	catan	catanf	catanl
acosh	acosh	acoshf	acoshl	cacosh	cacoshf	cacoshl
asinh	asinh	asinhf	asinhl	casinh	casinhf	casinhl
atanh	atanh	atanhf	atanhl	catanh	catanhf	catanhl
cos	cos	cosf	cosl	ccos	ccosf	ccosl
sin	sin	sinf	sinl	csin	csinf	csinl
tan	tan	tanf	tanl	ctan	ctanf	ctanl
cosh	cosh	coshf	coshl	ccosh	ccoshf	ccoshl
sinh	sinh	sinhf	sinhl	csinh	csinhf	csinhl
tanh	tanh	tanhf	tanhl	ctanh	ctanhf	ctanhl
exp	exp	expf	expl	cexp	cexpf	cexpl
log	log	logf	logl	clog	clogf	clogl
pow	pow	powf	powl	cpow	cpowf	cpowl
sqrt	sqrt	sqrtf	sqrtl	csqrt	csqrtf	csqrtl
fabs	fabs	fabsf	fabsl	cabs	cabsf	cabsl

型総称マクロ2

マクロ名に対応する関数がmath.hにだけあるもの(名前のみ示す)。

atan2	cbrt	ceil	copysign	erf	erfc	exp2	expm1
fdim	floor	fma	fmax	fmin	fmod	frexp	hypot
ilogb	ldexp	lgamma	llrint	llround	log10	log1p	log2
logb	lrint	lround	nearbyint	nextafter	nexttoward	remainder	remquo
rint	round	scalbn	scalbln	tgamma	trunc		

例

型総称マクロatan2に対応してatan2, atan2f, atan2lが呼び出される。

型総称マクロ3

マクロ名に対応する関数がcomplex.hにだけあるもの(名前のみ示す)

carg	conj	creal	cimag	cproj

例

型総称マクロcargに対応してcarg, cargf, carglが呼び出される。

L22 time.h 日付および時間 標準ライブラリの要約22

time.hは日付と時間を処理する機能をサポートします。「154 時間処理1」(p. 229)に関連説明があります。

型およびマクロ

NULL	空ポインタ定数（複数ヘッダで定義）
CLOCKS_PER_SEC	[clock関数戻り値÷CLOCKS_PER_SEC=秒数]となる定数
size_t型	sizeof演算子の結果を表現できる符号なし整数型（複数ヘッダで定義）
clock_t型	clock()関数が返す値を表現できる型
time_t型	time関数が返す値（時刻）を表すことができる型
tm型	時刻の要素（月、日、時、分、秒など）を保持する構造体（複数ヘッダで定義）

関数

```
char *asctime(const struct tm *timeptr);
```

時刻情報を文字列に変換して、そのポインタを返す

```
clock_t clock(void);
```

プログラム実行開始からの経過時間（プロセッサ時間）を返す

```
char *ctime(const time_t *timer);
```

timerが指す暦時刻を文字列形式の地方時に変換し、そのポインタを返す

```
double difftime(time_t time1, time_t time0);
```

time1とtime2との差を秒単位で返す

```
struct tm *gmtime(const time_t *timer);
```

暦時刻をtm構造体型の協定世界時（UTC）に変換する

```
struct tm *localtime(const time_t *timer);
```

暦時刻を、地方時で表した要素別の時刻に変換して、そのポインタを返す

```
time_t mktime(struct tm *timeptr);
```

timeptrの指すtm型構造体のデータを暦時刻（time_t型）に変換して返す

```
size_t strftime(char * restrict s, size_t maxsize, const char * restrict format,
const struct tm * restrict timeptr);
```

timeptrが指す日付・時刻情報をformatにしたがって書式化してsに入れる

```
time_t time(time_t *timer);
```

現在の暦時刻を返す。同じものを*timerにも入れる

L23 wchar.h 多バイト文字およびワイド文字処理 標準ライブラリの要約23

wchar.hは多バイト文字とワイド文字に関連する多くの機能をサポートします。このヘッダファイルはC89(追補1)で追加されました。

型およびマクロ

wchar_t型	ワイド文字を表現できる整数型(複数ヘッダで定義)
size_t型	sizeof演算子の結果を表現できる符号なし整数型(複数ヘッダで定義)
mbstate_t型	多バイト文字の並びとワイド文字の並びの間の変換状態情報を保持することができる型
wint_t型	拡張文字集合の全コードとWEOFを保持することができる整数型(複数ヘッダで定義)
tm型	時刻の要素(月、日、時、分、秒など)を保持する構造体(複数ヘッダで定義)
NULL	空ポインタ定数(複数ヘッダで定義)
WCHAR_MIN	wchar_t型の最小値(複数ヘッダで定義)
WCHAR_MAX	wchar_t型の最大値(複数ヘッダで定義)
WEOF	ファイルの終わりを示すワイド文字版のEOF(複数ヘッダで定義)

関数

wint_t **btowc**(int c);

バイト文字cのワイド文字表現を返す

wint_t **fgetwc**(FILE *stream);

fgetc()のワイド文字対応関数

wchar_t ***fgetws**(wchar_t * restrict s, int n, FILE * restrict stream);

fgets()のワイド文字対応関数

wint_t **fputwc**(wchar_t c, FILE *stream);

fputc()のワイド文字対応関数

int **fputws**(const wchar_t * restrict s, FILE * restrict stream);

fputs()のワイド文字対応関数

int **fwide**(FILE *stream, int mode);

mode値(0より大、0より小、0)によって指定ストリームの用途をワイド文字用、バイト処理用、現状維持、に設定する

int **fwprintf**(FILE * restrict stream, const wchar_t * restrict format, ...);

fprintf()のワイド文字対応関数

int **fwscanf**(FILE * restrict stream, const wchar_t * restrict format, ...);

fscanf()のワイド文字対応関数

wint_t **getwc**(FILE *stream);

getc()のワイド文字対応関数。マクロ実装されることがある

wint_t **getwchar**(void);

getchar()のワイド文字対応関数

```
size_t mbrlen(const char * restrict s, size_t n, mbstate_t * restrict ps);
```

mblen()関数に追加の引数psをもたせ、関数型をsize_tにしたものである。*psが保持しているシフト状態を反映して動作し、最後に*psをその時点のシフト状態に更新する。不正のときは特殊値を返す

```
size_t mbrtowc(wchar_t * restrict pwc, const char * restrict s, size_t n, mbstate_t * restrict ps);
```

mbtowc()関数に追加の引数psをもたせ、関数型をsize_tにしたものである。*psが保持しているシフト状態を反映して動作し、最後に*psをその時点のシフト状態に更新する。不正のときは特殊値を返す

```
int mbsinit(const mbstate_t *ps);
```

*psで示す値が非シフト状態(初期状態)なら真を返す。注：多バイト文字(複数バイトからなる)をワイド文字に変換すると、変換開始でシフト状態になり、変換終了で非シフト状態に戻る

```
size_t mbsrtowcs(wchar_t * restrict dst, const char ** restrict src, size_t len, mbstate_t * restrict ps);
```

mbstowcs()関数に追加の引数psをもたせ、引数srcをchar **型にしたものである。*psが保持しているシフト状態を反映して動作し、最後に*psをその時点のシフト状態に更新する。多バイト文字列*srcをワイド文字列に変換し、len個以下の文字をdstに格納する。*srcは次に変換される多バイト列を指すように更新する。格納したワイド文字数を返す。

```
wint_t putwc(wchar_t c, FILE *stream);
```

putc()のワイド文字対応関数。マクロ実装されることがある

```
wint_t putwchar(wchar_t c);
```

putchar()のワイド文字対応関数

```
int swprintf(wchar_t * restrict s, size_t n, const wchar_t * restrict format, ...);
```

sprintf()のワイド文字対応関数。ただし書き込まれるワイド文字数はnを超えない

```
int swscanf(const wchar_t * restrict s, const wchar_t * restrict format, ...);
```

sscanf()のワイド文字対応関数

```
wint_t ungetwc(wint_t c, FILE *stream);
```

ungetc()のワイド文字対応関数

```
int vfwprintf(FILE * restrict stream, const wchar_t * restrict format, va_list arg);
```

vfprintf()のワイド文字対応関数

```
int vfwscanf(FILE * restrict stream, const wchar_t * restrict format, va_list arg);  C99
```

vfscanf()のワイド文字対応関数

```
int vswprintf(wchar_t * restrict s, size_t n, const wchar_t * restrict format, va_list arg);
```

vsprintf()のワイド文字対応関数

```
int vswscanf(const wchar_t * restrict s, const wchar_t * restrict format, va_list arg);  C99
```

vsscanf()のワイド文字対応関数

```
int vwprintf(const wchar_t * restrict format, va_list arg);
```

vprintf()のワイド文字対応関数

```
int vwscanf(const wchar_t * restrict format, va_list arg);
```
C99

vscanf()のワイド文字対応関数

```
size_t wcrtomb(char * restrict s, wchar_t wc, mbstate_t * restrict ps);
```

wctomb()関数に追加の引数psをもたせ、関数型をsize_tにしたものである。*psが保持しているシフト状態を反映して動作し、最後に*psをその時点のシフト状態に更新する。不正のときは特殊値を返す

```
wchar_t *wcscat(wchar_t * restrict s1, const wchar_t * restrict s2);
```

strcat()のワイド文字対応関数

```
wchar_t *wcschr(const wchar_t *s, wchar_t c);
```

strchr()のワイド文字対応関数

```
int wcscmp(const wchar_t *s1, const wchar_t *s2);
```

strcmp()のワイド文字対応関数

```
int wcscoll(const wchar_t *s1, const wchar_t *s2);
```

strcoll()のワイド文字対応関数。地域仕様にしたがったワイド文字列比較

```
wchar_t *wcscpy(wchar_t * restrict s1, const wchar_t * restrict s2);
```

strcpy()のワイド文字対応関数

```
size_t wcscspn(const wchar_t *s1, const wchar_t *s2);
```

strcspn()のワイド文字対応関数

```
size_t wcsftime(wchar_t * restrict s, size_t maxsize, const wchar_t * restrict
format, const struct tm * restrict timeptr);
```

strftime()のワイド文字対応関数

```
size_t wcslen(const wchar_t *s);
```

strlen()のワイド文字対応関数

```
wchar_t *wcsncat(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);
```

strncat()のワイド文字対応関数

```
int wcsncmp(const wchar_t *s1, const wchar_t *s2, size_t n);
```

strncmp()のワイド文字対応関数

```
wchar_t *wcsncpy(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);
```

strncpy()のワイド文字対応関数

```
wchar_t *wcspbrk(const wchar_t *s1, const wchar_t *s2);
```

strpbrk()のワイド文字対応関数

```
wchar_t *wcsrchr(const wchar_t *s, wchar_t c);
```

strrchr()のワイド文字対応関数

```
size_t wcsrtombs(char * restrict dst, const wchar_t ** restrict src, size_t len,
mbstate_t * restrict ps);
```

wcstombs()関数に追加の引数psをもたせ、引数srcをchar **型にしたものである。*psが保持しているシフト状態を反映して動作し、最後に*psをその時点のシフト状態に更新する。ワイド文字列*srcを多バイト文字列に変換して、len個以下の文字をdstに格納する。*srcは次に変換されるワイド文字列を指すように更新する。格納したバイト数を返す

```
size_t wcsspn(const wchar_t *s1, const wchar_t *s2);
```

strspn()のワイド文字対応関数

```
wchar_t *wcsstr(const wchar_t *s1, const wchar_t *s2);
```

strstr()のワイド文字対応関数

```
double wcstod(const wchar_t * restrict nptr, wchar_t ** restrict endptr);
float wcstof(const wchar_t * restrict nptr, wchar_t ** restrict endptr);  C99
long double wcstold(const wchar_t * restrict nptr, wchar_t ** restrict endptr);  C99
```

strtod(),strtof(),strtold()のワイド文字対応関数

```
wchar_t *wcstok(wchar_t * restrict s1, const wchar_t * restrict s2, wchar_t **
restrict ptr);
```

strtok()のワイド文字対応関数

```
long int wcstol(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int
base);
long long int wcstoll(const wchar_t * restrict nptr, wchar_t ** restrict endptr,
int base);  C99
unsigned long int wcstoul(const wchar_t * restrict nptr, wchar_t ** restrict
endptr, int base);
unsigned long long int wcstoull(const wchar_t * restrict nptr, wchar_t ** restrict
endptr, int base);  C99
```

strtol(),strtoll(),strtoul(),strtoull()のワイド文字対応関数

```
size_t wcsxfrm(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);
```

strxfrm()のワイド文字対応関数。文字列を地域仕様にもとづいて変換する

```
int wctob(wint_t c);
```

ワイド文字cに対応する1バイト文字表現を返す。1バイトで表現できないときはEOFを返す

```
wchar_t *wmemchr(const wchar_t *s, wchar_t c, size_t n);
```

memchr()のワイド文字対応関数

```
int wmemcmp(const wchar_t * s1, const wchar_t * s2, size_t n);
```

memcmp()のワイド文字対応関数

```
wchar_t *wmemcpy(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);
```

memcpy()のワイド文字対応関数

```
wchar_t *wmemmove(wchar_t *s1, const wchar_t *s2, size_t n);
```

memmove()のワイド文字対応関数

```
wchar_t *wmemset(wchar_t *s, wchar_t c, size_t n);
```

memset()のワイド文字対応関数

```
int wprintf(const wchar_t * restrict format, ...);
```

printf()のワイド文字対応関数

```
int wscanf(const wchar_t * restrict format, ...);
```

scanf()のワイド文字対応関数

L24 wctype.h ワイド文字の分類および変換 標準ライブラリの要約24

wctype.hはワイド文字の文字種分類と変換の機能をサポートします。このヘッダファイルはC89(追補1)で追加されました。

型およびマクロ

wint_t型	拡張文字集合の全コードとWEOFを保持することができる整数型(複数ヘッダで定義)
wctrans_t型	ロケール固有の文字写像を表現する値を保持できる型
wctype_t型	ロケール固有の文字種分類を表現する値を保持できる型
WEOF	ファイルの終わりを示すワイド文字版のEOF(複数ヘッダで定義)

関数

```
int iswalnum(wint_t wc);
```

isalnum()のワイド文字対応関数

```
int iswalpha(wint_t wc);
```

isalpha()のワイド文字対応関数

```
int iswblank(wint_t wc);
```
C99

isblank()のワイド文字対応関数

```
int iswcntrl(wint_t wc);
```

iscntrl()のワイド文字対応関数

```
int iswctype(wint_t wc, wctype_t desc);
```

ワイド文字wcがdescで指定する文字種別に属すれば真を返す。iswctype(wc, wctype("alnum"))はiswalnum(wc)と等価である

```
int iswdigit(wint_t wc);
```

isdigit()のワイド文字対応関数

```
int iswgraph(wint_t wc);
```

isgraph()のワイド文字対応関数

```
int iswlower(wint_t wc);
```

islower()のワイド文字対応関数

```
int iswprint(wint_t wc);
```

isprint()のワイド文字対応関数

```
int iswpunct(wint_t wc);
```

ispunct()のワイド文字対応関数

```
int iswspace(wint_t wc);
```

isspace()のワイド文字対応関数

```
int iswupper(wint_t wc);
```

isupper()のワイド文字対応関数

```
int iswxdigit(wint_t wc);
```

isxdigit()のワイド文字対応関数

```
wint_t towctrans(wint_t wc, wctrans_t desc);
```

ワイド文字wcをdescで指定する指示にしたがって写像(文字変換)する。towctrans(wc, wctrans("tolower"))はtowlower(wc)と同じである

```
wint_t towlower(wint_t wc);
```

tolower()のワイド文字対応関数

```
wint_t towupper(wint_t wc);
```

toupper()のワイド文字対応関数

```
wctrans_t wctrans(const char *property);
```

文字列propertyの写像(文字変換)指示に対応するwctrans_t型の値を返す。戻り値はtowctrans()関数の第2引数に利用することで実際の変換ができる。propertyでは少なくとも、"tolower", "toupper"の指定は可能である

```
wctype_t wctype(const char *property);
```

文字列propertyで指定するワイド文字種別に対応するwctype_t型の値を返す。戻り値はiswctype()関数の第2引数に利用することで実際の判定ができる。propertyでは少なくとも、"alnum", "alpha", "blank", "cntrl", "digit", "graph", "lower", "print", "punct", "space", "upper", "xdigit"の指定は可能である

第20章

C Quick Reference 標準ライブラリ関数一覧

概要

主要な標準ライブラリ関数(C89規格)の仕様と簡単な用例をまとめました。最新規格であるC99では膨大な関数が追加になっていますが、実用的な用途においてはC89規格の関数で十分です。ここでの関数説明は簡潔に内容を紹介するのが目的であり、文法的に完全な説明をしているわけではありません。完全な文法を知りたいときは、製品のマニュアルやJIS C規格書をご覧ください。

関数形式の中で「...」は可変個の引数を意味します。たとえば可変個引数をとるprintf()関数のフォーマットは次のとおりです。

```
int printf(const char *format, ...)
```

またC99規格が登場して、関数引数に型修飾子restrict修飾子がつくものが多くあります。たとえば、

```
char *strcpy(char * restrict s1, const char * restrict s2);
```

のようになります。規格では「restrict修飾子を削除しても、その意味(すなわち、目に見える動作)は変わらない」とされています。この章では、簡潔表現のためにrestrictのない書式を示します。

abort

形式
```
#include <stdlib.h>
void abort(void);
```
戻値 なし

説明 プログラムを意図的に異常終了させる(シグナルSIGABRTを発生する)。このとき処理系定義の形式で失敗終了状態をホスト環境に返す。後始末処理をどのように行なうかは処理系定義。→signal()関数(p. 309)

用例
```
if (条件) abort();        // 何かの条件で中断する
```

abs

形式
```
#include <stdlib.h>
int abs(int n);
```
戻値 nの絶対値

説明 数値nの絶対値を返す。

用例
```
n = abs(-12);     // nは12になる
```

acos

形式
```
#include <math.h>
double acos(double x);
```
戻値 計算結果

説明 アークコサイン値(逆余弦)を返す。

用例
```
d = acos(0.866) / 3.14159 * 180.0;  // dは30.002936。コサイン30度は0.866
```

asctime

形式
```
#include <time.h>
char *asctime(const struct tm *tmptr);
```
戻値 変換した文字列へのポインタ

説明 tmptrの指す構造体にある時間情報を表示可能文字列に変換する。変換した文字列の最後には'¥n'が入っている。tm構造体はtime.hの中で宣言されている。→time()関数(p. 318), localtime()関数(p. 299), gmtime()関数(p. 295)

用例 →「現在時刻を表示する」(p. 231)

asin

形式
```
#include <math.h>
double asin(double x);
```
戻値 計算結果

説明 アークサイン値（逆正弦）を返す。

用例
```
d = asin(0.5) / 3.14159 * 180.0;     // dは30.000025。サイン30度は0.5
```

assert

形式
```
#include <assert.h>
void assert(int expression);
void assert(スカラー型 expression);  ── C99の書式
```
戻値 なし

説明 条件expressionが偽（0）のとき、診断メッセージを出力し、abort()関数を呼び出して処理を中断する。出力メッセージには「式、ファイル名、行番号」が含まれる。C99では関数名も含まれる。assert()はマクロで実現されており、デバッグのときに使用する。#include <assert.h>の前で「#define NDEBUG」が定義されているとassert()は無効になる。このため完成プログラムにassert()記述を埋め残すことができる。

用例
```
/* astst.c */
#include <stdio.h>
//#define NDEBUG       // コメントを外すとassert()が無効になる
#include <assert.h>
int main(int argc, char *argv[])
{
    printf("開始します¥n");
    assert(argc!=1);
    printf("終了します¥n");
    return 0;
}
```

実行結果：Visual C++で引数なし実行した場合
```
開始します
Assertion failed: argc!=1, file cq_081.c, line 9

This application has requested the Runtime to terminate it in an unusual way.
Please contact the application's support team for more information.
```

atan

形式
```
#include <math.h>
double atan(double x);
```
戻値 計算結果

説明 アークタンジェント値（逆正接）を返す。

用例
```
d = atan(0.577) / 3.14159 * 180.0;    // dは29.984971。タンジェント30度は0.577
```

atan2

形式
```
#include <math.h>
double atan2(double y, double x);
```
戻値 計算結果

説明 y/xのアークタンジェント値（y/xの逆正接）を返す。直交座標位置の角度表現。

用例
```
d = atan2(0.577, 1.0) / 3.14159 * 180.0;
                         // dは29.984971。x=1, y=0.577は極座標で約30度
```

atexit

形式 `#include <stdlib.h>`
`int atexit(void (*func)(void));`

戻値 正常時：0　エラー時：非0

説明 プログラムの正常終了時に自動実行される関数func()を登録する。関数func()は引数も戻り値もない関数でなければならない。複数の関数（最低32個）を登録できる。登録と逆順に実行される。

用例

```
#include <stdio.h>
#include <stdlib.h>
void fnc1(void)
{
    puts("終了します。Enterキーを押してください。");
    (void)getchar();
}

void fnc2(void)
{
    puts("お仕事ごくろうさまでした。");
}

int main(void)
{
    atexit(fnc1);
    atexit(fnc2);
    puts("end");
    return 0;
}
```

実行結果

```
end
お仕事ごくろうさまでした。
終了します。Enterキーを押してください。
```

atof

形式 `#include <stdlib.h>`
`double atof(const char *s);`

戻値 変換された浮動小数点数値

説明 文字列sの先頭部分をdouble型の値に変換する。変換できないときの処理は未定義だが通常は0.0になる。数値変換に関してはstrtod(s, (char **)NULL)と同じである。

用例

```
ddt = atof("12.34");          // 12.34
ddt = atof("-12.34");         // -12.34
ddt = atof("  12.34XYZ");     // 12.34
ddt = atof("XYZ");            // 0.0
```

atoi

形式 `#include <stdlib.h>`
`int atoi(const char *s);`

戻値 変換されたint値

説明 文字列sの先頭部分をint型の値に変換する。変換できないときの処理は未定義だが通常は0になる。数値変換に関してはは(int)strtol(s, (char **)NULL, 10)と同じである。

用例

```
idt = atoi("1234");           // 1234
idt = atoi("-1234");          // -1234
idt = atoi("  1234XYZ");      // 1234
idt = atoi("XYZ");            // 0
```

atol

形式
```
#include <stdlib.h>
long int atol(const char *s);
```

戻値 変換されたlong値

説明 文字列sの先頭部分をlong型の値に変換する。変換できないときの処理は未定義だが通常は0Lになる。数値変換に関してはstrtol(s, (char **)NULL, 10)と同じである。

用例
```
ldt = atol("1234");          // 1234
ldt = atol("-1234");         // -1234
ldt = atol("  1234XYZ");     // 1234
ldt = atol("XYZ");           // 0
```

bsearch

形式
```
#include <stdlib.h>
void *bsearch(const void *key, const void *base, size_t n, size_t size,
                               int (*cmp)(const void *a, const void *b));
```

戻値 正常時：マッチした項目へのポインタ　マッチしないとき：NULL

説明 バイナリ・サーチ（二分探索）を行なう。base[0]～base[n-1]の中からkeyで示すデータを探す。ひとつの要素のバイト長をsizeで指定する。配列baseの中身は昇順にソート済みでなければならない。関数cmp()はふたつの引数の示す値を比較し、a>bなら正、a=bなら0、a<bなら負、を返すものとする。この関数はbsearch()関数内で自動的に呼ばれ、引数aにはkeyへのポインタ、引数bには配列要素へのポインタが渡される。

用例
```
#include <stdio.h>
#include <stdlib.h>
int intcmp(const void *a, const void *b)     // int昇順比較関数
{
    return *(int *)a - *(int *)b;
}

int main(void)
{
    int entry[] = {34, 54, 26, 52, 87, 37, 55};
    int n, nbr, *p;

    n = sizeof(entry)/sizeof(entry[0]);      // データ個数
    qsort(entry, n, sizeof(int), intcmp);    // ソートしておく
    nbr = 52;
    p = (int *)bsearch(&nbr, entry, n, sizeof(int), intcmp);
    if (p == NULL) printf("登録されていない¥n");
    else printf("%dは登録されている¥n", *p);    // 出力：52は登録されている
    return 0;
}
```

calloc

形式
```
#include <stdlib.h>
void *calloc(size_t n, size_t size);
```

戻値 正常時：割り当てたメモリへのポインタ、エラー時：NULL

説明 sizeバイト×n個分のメモリ領域を確保し、それへのポインタを返す。確保した領域は0に初期化される。確保領域は適切に境界調整されている。

用例
```
int *p;
p = (int *)calloc(500, sizeof(int));     // int変数500個分のメモリを確保
if (p == NULL) {
    printf("out of memory"); exit(1);    // メモリ確保に失敗した
}
```

ceil

形式 `#include <math.h>`
`double ceil(double x);`

戻値 計算結果

説明 数値を切り上げる(ceiling：天井)。x以上の最小の整数値をdouble型で返す。

用例
```
d = ceil(12.11);        // dは13になる
d = ceil(-5.68);        // dは -5になる( -5.68より大きい最小の整数)
```

clearerr

形式 `#include <stdio.h>`
`void clearerr(FILE *stream);`

戻値 なし

説明 streamが指すストリームのファイル終了表示子と、エラー表示子をクリアする。

用例
```
if (ferror(fp)) {       // エラー発生なら
    (適切なエラー処理をして)
    clearerr(fp);       // エラー表示子をクリアする
}
```

clock

形式 `#include <time.h>`
`clock_t clock(void);`

戻値 正常時：経過時間　エラー時：(clock_t)(-1)　――**clock_t**型にキャストした-1

説明 プログラム実行開始からの経過時間を得る。clock()関数は微小時間単位(1秒より短い)の戻り値を返す。これをCLOCKS_PER_SEC定数で割ると秒単位にすることができる。

用例 1秒間隔で数字を表示する
```
#include <stdio.h>
#include <time.h>
int main(void)
{
    clock_t t1, t2;
    int i;
    for (i=1; i<=5; i++) {
        t1 = clock();
        while (1) {
            t2 = clock();
            if ((t2 - t1) / CLOCKS_PER_SEC >= 1) break;
        }
        printf("%d¥n", i);          // 1秒間隔で表示
    }
    return 0;
}
```

cos

形式 `#include <math.h>`
`double cos(double x);`

戻値 計算結果

説明 コサイン値(余弦)を返す。

用例
```
d = cos(30*3.14159/180.0);          // dは0.866026。コサイン30度
```

cosh

形式
```
#include <math.h>
double cosh(double x);
```

戻値 計算結果

説明 ハイパボリックコサイン値（双曲線余弦）を返す。

ctime

形式
```
#include <time.h>
char *ctime(const time_t *t);
```

戻値 変換した文字列へのポインタ

説明 time()関数で取得した暦時刻tを現地時間（日本時間）の表示可能文字列に変換する。変換した文字列の最後には'¥n'が入っている。

用例 →「現在時刻を表示する」(p. 231)

difftime

形式
```
#include <time.h>
double difftime(time_t t1, time_t t2);
```

戻値 得られた時間差（秒単位）

説明 t1-t2の結果を秒単位で計算する。

用例 時間間隔を測定。clock()関数で示したプログラム例をdifftime()で記述する

```
#include <stdio.h>
#include <time.h>
int main(void)
{
    time_t t1, t2;
    int i;

    for (i=1; i<=5; i++) {
        time(&t1);
        while (1) {
            time(&t2);
            if (difftime(t2, t1) >= 1.0 ) break;
        }
        printf("%d¥n", i);          // 1秒間隔で表示
    }
    return 0;
}
```

div

形式
```
#include <stdlib.h>
div_t div(int a, int b);
```

戻値 計算結果を入れた構造体

説明 a÷bを計算し、その商（=quotient）をdiv_t型のメンバquotに、剰余（=remainder）をdiv_t型のメンバremに入れる。

用例
```
div_t d;
d = div(20, 6);
printf("商=%d 余り=%d¥n", d.quot, d.rem);     // 出力：商=3 余り=2
```

exit

形式
```
#include <stdlib.h>
void exit(int status);
```

戻値 なし

説明 atexit()で登録した関数があればそれを実行し、オープンされているすべてのストリームを閉じ、プログラムを終了する。数値statusを終了コードとして親プロセスに返す。正常終了にexit(0)を、異常終了にexit(1)を用いることが多い。また次の標準値がstdlib.hの中で定義されている。

EXIT_SUCCESS ──正常終了を示す値

EXIT_FAILURE ──異常終了を示す値

用例
```
exit(EXIT_FAILURE);   // 例1
exit(1);              // 例2
```

exp

形式
```
#include <math.h>
double exp(double x);
```

戻値 計算結果

説明 自然対数の底e(=2.7182…)の累乗を返す。

用例
```
d = exp(1.0);     // dは2.718282。自然対数の底そのもの
d = exp(2.0);     // dは7.389056。2.718282の2.0乗
```

fabs

形式
```
#include <math.h>
double fabs(double x);
```

戻値 計算結果

説明 double型データxの絶対値を返す。

用例
```
d = fabs(-23.4); // dは23.4になる
```

fclose

形式
```
#include <stdio.h>
int fclose(FILE *stream);
```

戻値 正常時：0　エラー時：EOF

説明 streamで示すファイルをクローズする。バッファリングされていてまだ未書き込みのデータはファイルに正しく書き込まれる。

用例
```
fclose(fp);
```

feof

形式
```
#include <stdio.h>
int feof(FILE *stream);
```

戻値 ファイル終了：非0　終了でない：0

説明 streamのファイル終了表示子をチェックする。streamがファイルの終わりに達しているかどうかを検査できる。

用例
```
if (feof(fp)) ～        // ファイル終了なら
```

ferror

形式
```
#include <stdio.h>
int ferror(FILE *stream);
```

戻値 エラーが発生している：非0　エラーはない：0

説明 streamのエラー表示子がセットされていること(=エラー発生)を検査する。

用例
```
if (ferror(fp)) {       // エラー発生なら
   (エラー処理をして)
    clearerr(fp);       // エラー表示子をクリアする
}
```

fflush

形式
```
#include <stdio.h>
int fflush(FILE *stream);
```
戻値 正常時：0　エラー時：EOF

説明 streamで示すストリームにまだ書き込まれていない出力データがあれば、それをファイルに書き込む。引数にNULLを指定したときはオープンされている全ファイルを対象にする。

用例
```
fflush(fp);          // ストリームの出力データを強制書き出しする
```

fgetc

形式
```
#include <stdio.h>
int fgetc(FILE *stream);
```
戻値 正常時：読み込んだ文字　ファイル終了時：EOF(ファイル終了表示子もセット)

説明 streamが指すストリームから1文字読み込む。文字はunsigned char型として取り込みint型に変換する。

用例
```
ch = fgetc(fp);          // ファイルfpから1文字読み込みchに入れる
```

fgetpos

形式
```
#include <stdio.h>
int fgetpos(FILE *stream, fpos_t *pos);
```
戻値 正常時：0　エラー時：非0(エラー番号をerrnoに格納)

説明 streamが指すストリームのファイル位置表示子情報を*posに格納する。その値はあとでfsetpos()関数で使用できる。fgetpos()とfsetpos()を用いることで、同じ位置から読み書きを再開できる。

用例 →「fgetposとfsetpos」(p. 216)

fgets

形式
```
#include <stdio.h>
char *fgets(char *s, int n, FILE *stream);
```
戻値 正常時：sを返す　ファイル終了またはエラー時：NULL

説明 streamが指すストリームからn-1文字まで、または改行まで、またはファイル終了までの文字列を読み込む。読み込んだ文字列の最後に'¥0'を付加する。最後の'¥0'を含めて最大n文字になる。改行文字もそのまま読み込む。

用例 sに最大256文字('¥0'も含めて)をファイルfpから読み込む
```
char s[256];
...
while (fgets(s, 256, fp) != NULL) {     // ファイル終了まで1行入力
    printf("%s", s);                    // それを画面出力
}
```

floor

形式
```
#include <math.h>
double floor(double x);
```
戻値 計算結果

説明 数値を切り下げる(floor：床)。x以下の最大の整数値をdouble型で返す。

用例
```
d = floor(12.11);       // dは12になる
d = floor(-5.68);       // dは -6になる(-5.68より小さい最大の整数)
```

fmod

形式
```
#include <math.h>
double fmod(double x, double y);
```
戻値 計算結果

説明 x/yの剰余(浮動小数点数剰余)を計算する。

用例
```
d = fmod(10.5, 3.2);         // dは0.9になる。10.5 - 3.2 × 3 = 0.9
```

fopen

形式
```
#include <stdio.h>
FILE *fopen(const char *filename, const char *mode);
```

戻値 正常時：オープンしたファイルへのポインタ　エラー時：NULL

説明 filenameで示すファイルを、modeで示すモードでオープンする。オープンモードの例 "r"：読み込み "w"：書き込み "a"：追加書き込み。詳細は「140 ファイル処理の手順1」(p. 202)参照。

用例
```
FILE *fp;
if ((fp=fopen("myfile.txt","r")) == NULL) {      // 読み込み用に開く
    printf("ファイルをオープンできません¥n");      // エラー表示
    exit(1);                                     // 打ち切り
}
```

fprintf

形式
```
#include <stdio.h>
int fprintf(FILE *stream, const char *format, ...);
```

戻値 正常時：出力した文字数　エラー時：負の値

説明 対応する引数の内容をformatで指定する書式にしたがってstreamが指すストリームに出力する。formatの記述方法については「133 書式つき出力2」(p. 186)を参照。

用例
```
fprintf(fp, "%d¥n", dt);      // ストリームfpに変数dtの値を10進数で出力
```

fputc

形式
```
#include <stdio.h>
int fputc(int ch, FILE *stream);
```

戻値 正常時：出力した文字　エラー時：エラー表示子をセットしEOFを返す

説明 文字chをstreamが指すストリームに出力する。

用例
```
fputc('a', fp);              // 文字'a'を出力
```

fputs

形式
```
#include <stdio.h>
int fputs(char *s, FILE *stream);
```

戻値 正常時：非負の値　エラー時：EOF

説明 文字列sをstreamが指すストリームに出力する。終端文字'¥0'は書き込まず、改行文字を付加することもない。

用例
```
fputs("abcd", fp);           // ストリームfpに文字列を出力
```

fread

形式
```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t n, FILE *stream);
```

戻値 読み込んだデータの個数
戻り値がnより小さいときはファイル終了かエラー発生である
ファイル終了またはエラー発生を区別するにはfeof(), ferror()関数を利用すればよい
sizeまたはnが0のときは読み込みは行なわず0を返す

説明 streamが指すストリームから「sizeバイト×n個」分のデータを配列ptrに読み込む。

用例 →「146 ブロック単位の読み書き」(p. 212)

free

形式
```
#include <stdlib.h>
void free(void *ptr);
```
戻値 なし

説明 malloc(), calloc(), realloc()の各関数によって確保された、ptrで示すメモリ領域を解放する。以前に確保されたポインタと一致しないときや、ptrで示す領域がすでに解放済みのときの動作は未定義。ptrがNULLの場合は何もしない。

用例
```
char *cp;
cp = (char *)malloc(500);     // 500バイト分メモリを確保する
...
free(cp);                     // それを解放する
```

freopen

形式
```
#include <stdio.h>
FILE *freopen(const char *filename, const char *mode, FILE *stream);
```
戻値 正常時：streamを返す　エラー時：NULL

説明 まずstreamで示すオープン済みのファイルをクローズする。次にfilenameで示すファイルをmodeで指定したモードでオープンしstreamが指すストリームに結びつける。オープンモードはfopen()と同じ。filenameがNULLのときは現在のファイルのmodeのみを変更する。

用例 stdoutをtst.txtファイルに割りつける。"string1"は画面に"string2"はtst.txtに書き込まれる
```
#include <stdio.h>
int main(void)
{
    printf("string1¥n");      // 画面に出力
    if (freopen("tst.txt", "w", stdout) == NULL) return 1;
    printf("string2¥n");      // tst.txtに出力
    return 0;
}
```

frexp

形式
```
#include <math.h>
double frexp(double value, int *p);
```
戻値 計算結果

説明 指数分解（FRaction EXPonential）をする。浮動小数点数を小数xと指数nに分解する。それぞれの値は、value=x×2^nとなる仮数部xと指数nである。仮数部xを戻り値とし、指数部nは*pに入る。xは0.0または、0.5以上、1.0未満の値をとる。

用例
```
double x;
int p;
x = frexp(4.0, &p);   // xは0.5にpは3になる。4.0 = 0.5 × 2³
```

fscanf

形式
```
#include <stdio.h>
int fscanf(FILE *stream, const char *format, ...);
```
戻値 正常時：入力データの個数　ファイル終了またはエラー時：EOF

説明 streamが指すストリームから書式formatにしたがい対応する引数にデータを読み込む。formatの記述方法については「136 書式つき入力2」（p. 192）を参照。

用例
```
int dt;
...
fscanf(fin, "%d", &dt);   // finから10進数をdtに読み込む
```

fseek

形式
```
#include <stdio.h>
int fseek(FILE *stream, long int offset, int whence);
```
戻値 正常時：0　エラー時：非0

説明 streamが指すストリームのファイル位置表示子をwhenceを基点としてoffsetバイト目の位置に移動する。offsetをマイナス値にすれば基点より先頭方向の指定になる。whence（移動の基点）の指定方法は以下のとおり。

SEEK_SET	先頭位置から
SEEK_CUR	現在位置から
SEEK_END	終端位置から

バイナリストリームの場合は指定したとおりに機能する。テキストストリームの場合は次の操作だけが安全である。(1)offsetに0を指定して移動する。(2)whenceにSEEK_SETを、offsetに「以前に取得したftell()関数の戻り値」を指定して移動する。

用例 →「147 読み書き位置の指定1」(p. 213)

fsetpos

形式
```
#include <stdio.h>
int fsetpos(FILE *stream, const fpos_t *pos);
```
戻値 正常時：0　エラー時：非0（エラー番号をerrnoに格納）

説明 streamが指すストリームのファイル位置表示子を*posでセットしなおす。*posの値はあらかじめfgetpos()関数で取得しておいたものを用いる。fgetpos()とfsetpos()を用いることで、同じ位置から読み書きを再開できる。

用例 →「fgetposとfsetpos」(p. 216)

ftell

形式
```
#include <stdio.h>
long int ftell(FILE *stream);
```
戻値 正常時：現在のファイル位置表示子　エラー時：-1L（エラー番号をerrnoに格納）

説明 streamが指すストリームの現在のファイル位置表示子を返す。この値はfseek()関数の引数として利用できる。

用例 →「147 読み書き位置の指定1」(p. 213)

fwrite

形式
```
#include <stdio.h>
size_t fwrite(const void *ptr, size_t size, size_t n, FILE *stream);
```
戻値 正しく書き込んだデータの個数
戻り値がnより小さいときはエラー発生と判断できる
sizeまたはnが0のときは書き込みは行なわず0を返す

説明 ptrの先頭から「sizeバイト×n個」分のデータをstreamが指すストリームに書き込む。

用例 →「146 ブロック単位の読み書き」(p. 212)

getc

形式
```
#include <stdio.h>
int getc(FILE *stream);
```
戻値 正常時：読み込んだ文字　ファイル終了時：EOF（ファイル終了表示子もセット）

説明 streamが指すストリームから1文字読み込む。機能はfgetc()関数と等価。getc()はマクロとして実現されることがある。

用例
```
ch = getc(fp);    // ファイルfpから1文字読み込みchに入れる
```

getchar

形式 **#include <stdio.h>**

int getchar(void);

戻値 正常時：読み込んだ文字　ファイル終了時／エラー時：EOF

ファイル終了時はファイル終了表示子を、エラー時はエラー表示子をセットする

説明 stdinから1文字を入力する。

用例
```
ch = getchar();  // 1文字読み込んでchに入れる
```

getenv

形式 **#include <stdlib.h>**

char *getenv(const char *env);

戻値 正常時：該当する文字列へのポインタ　環境変数が見つからないとき：NULL

説明 envで示すホスト環境の管理する環境変数の値を得る。

用例
```
char *p;
p = getenv("PATH");
if (p != NULL) printf("%s¥n", p);    // 環境変数PATHの内容を表示する
```

gets

形式 **#include <stdio.h>**

char *gets(char *s);

戻値 正常時：ポインタs　ファイル終了／エラー時：NULL

説明 stdinから改行文字まで、またはファイルの終わりまでの文字列を入力しsに格納する。改行文字は捨てられる。

用例
```
gets(s);  // 1行入力してsに入れる
```

gmtime

形式 **#include <time.h>**

struct tm *gmtime(const time_t *t);

戻値 変換した時間情報へのポインタ　エラー時：NULL

説明 暦時刻をtm構造体型の協定世界時（UTC）に変換する。

用例 →「現在時刻を表示する」（p. 231）

isalnum

形式 **#include <ctype.h>**

int isalnum(int ch);

戻値 英数字のとき：非0　そうでないとき：0

説明 文字chが英数字（A～Z, a～z, 0～9）なら真を返す（Cロケールの場合）。

用例
```
for (c=0; c<=127; c++) { if (isalnum(c)) putchar(c); }
結果：0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
```

isalpha

形式 **#include <ctype.h>**

int isalpha(int ch);

戻値 英文字のとき：非0　そうでないとき：0

説明 文字chが英文字（A～Z, a～z）なら真を返す（Cロケールの場合）。

用例
```
for (c=0; c<=127; c++) { if (isalpha(c)) putchar(c); }
結果：ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
```

isdigit

形式
```
#include <ctype.h>
int isdigit(int ch);
```
戻値 10進数字のとき：非0　そうでないとき：0

説明 文字chが10進数字（0〜9）なら真を返す。

用例
```
for (c=0; c<=127; c++) { if (isdigit(c)) putchar(c); }
結果：0123456789
```

isgraph

形式
```
#include <ctype.h>
int isgraph(int ch);
```
戻値 空白以外の表示可能文字のとき：非0　そうでないとき：0

説明 文字chが空白を除く表示可能文字。

用例
```
for (c=0; c<=127; c++) { if (isgraph(c)) putchar(c); }
結果：!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[¥]^_`abcdefgh
ijklmnopqrstuvwxyz{|}~
```

islower

形式
```
#include <ctype.h>
int islower(int ch);
```
戻値 英小文字のとき：非0　そうでないとき：0

説明 文字chが英小文字（a〜z）なら真を返す（Cロケールの場合）。

用例
```
for (c=0; c<=127; c++) { if (islower(c)) putchar(c); }
結果：abcdefghijklmnopqrstuvwxyz
```

isprint

形式
```
#include <ctype.h>
int isprint(int ch);
```
戻値 表示可能文字のとき：非0　そうでないとき：0

説明 文字chが表示可能文字なら真を返す。

用例
```
for (c=0; c<=127; c++) { if (isprint(c)) putchar(c); }
結果：!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[¥]^_`abcdefgh
ijklmnopqrstuvwxyz{|}~
```

ispunct

形式
```
#include <ctype.h>
int ispunct(int ch);
```
戻値 区切り文字のとき：非0　そうでないとき：0

説明 文字chが区切り文字（punctuation character）なら真を返す（Cロケールの場合）。区切り文字とは空白、英数字以外の表示可能文字。

用例
```
for (c=0; c<=127; c++) { if (ispunct(c)) putchar(c); }
結果：!"#$%&'()*+,-./:;<=>?@[¥]^_`{|}~
```

isspace

形式
```
#include <ctype.h>
int isspace(int ch);
```

戻値 空白類文字のとき：非0　そうでないとき：0

説明 文字chが空白、タブ、復帰、改行、垂直タブ、フォームフィード（0x09～0x0D, 0x20）なら真を返す（Cロケールの場合）。

用例
```
for (c=0; c<=127; c++) { if (isspace(c)) printf("%02X ", c); }
結果：09 0A 0B 0C 0D 20
```

isupper

形式
```
#include <ctype.h>
int isupper(int ch);
```

戻値 大文字のとき：非0　そうでないとき：0

説明 文字chが大文字（A～Z）なら真を返す（Cロケールの場合）。

用例
```
for (c=0; c<=127; c++) { if (isupper(c)) putchar(c); }
結果：ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

isxdigit

形式
```
#include <ctype.h>
int isxdigit(int ch);
```

戻値 16進文字のとき：非0　そうでないとき：0

説明 文字chが16進文字（0～9, A～F, a～f）なら真を返す。

用例
```
for (c=0; c<=127; c++) { if (isxdigit(c)) putchar(c); }
結果：0123456789ABCDEFabcdef
```

labs

形式
```
#include <stdlib.h>
long int labs(long int n);
```

戻値 nの絶対値

説明 long型データnの絶対値を返す。

用例
```
long d =  -987654321;
printf("%ld¥n", labs(d));     // 出力：987654321
```

ldexp

形式
```
#include <math.h>
double ldexp(double x, int n);
```

戻値 計算結果

説明 $x \times 2^n$という値を返す。2^n乗という指数（EXPonential）にx倍という重み（LoaD）をつける。

用例
```
d = ldexp(0.5, 3);     // dの値は4.0になる
```

ldiv

形式
```
#include <stdlib.h>
ldiv_t ldiv(long int a, long int b);
```

戻値 計算結果を入れた構造体を返す

説明 long型の値でa÷bを計算し、その商（=quotient）をldiv_t型のメンバquotに、剰余（=remainder）をldiv_t型のメンバremに入れる。

用例
```
ldiv_t d;
d = ldiv(7654321, 7654);
printf("商=%ld 余り=%ld¥n", d.quot, d.rem);  // 出力：商=1000 余り=321
```

localeconv

形式
```
#include <locale.h>
struct lconv *localeconv(void);
```

戻値 lconv型構造体へのポインタ

説明 地域で使われる数字や通貨記号などの表現方法に関する現在のロケール情報を構造体lconvに設定する。lconv構造体の内容は次のとおり。表の中で「C地域値」は"C"ロケールでの設定値を示す。CHAR_MAXは未設定（無効設定）であることを示す。これらの内容を見るとCロケールが最小機能設定であることが分かる。

lconv構造体のメンバ（C89仕様）

メンバ	説明	C地域値
char *decimal_point;	小数点文字	"."
char *thousands_sep;	非金額の1000倍ごと区切文字	""
char *grouping;	非金額のグループ桁数	""
char *int_curr_symbol;	3文字の国際通貨記号 金額との区切り文字が後続する場合もある	""
char *currency_symbol;	通貨記号	""
char *mon_decimal_point;	金額の小数点文字	""
char *mon_thousands_sep;	金額の1000倍ごと区切文字	""
char *mon_grouping;	金額のグループ桁数	""
char *positive_sign;	正金額の符号文字	""
char *negative_sign;	負金額の符号文字	""
char int_frac_digits;	国際通貨金額の小数点以下桁数	CHAR_MAX
char frac_digits;	非国際通貨金額の小数点以下桁数	CHAR_MAX
char p_cs_precedes;	通貨記号を正金額の前or後につける指定	CHAR_MAX
char p_sep_by_space;	通貨記号と正金額を空白で区切るかどうか	CHAR_MAX
char n_cs_precedes;	負金額に対するp_cs_precedes的指定	CHAR_MAX
char n_sep_by_space;	負金額に対するp_sep_by_space的指定	CHAR_MAX
char p_sign_posn;	positive_signをつける位置	CHAR_MAX
char n_sign_posn;	negative_signをつける位置	CHAR_MAX

注：C99では以下の追加項目がある（すべてchar型）

int_p_cs_precedes　int_p_sep_by_space　int_p_sign_posn
int_n_cs_precedes　int_n_sep_by_space　int_n_sign_posn

用例 次のプログラムはVisual C++のみで動作を確認している

```
#include <stdio.h>
#include <locale.h>
int main(void)
{
    struct lconv *lc;
    setlocale(LC_ALL, "");          // 地域性を日本にする
    lc = localeconv();
    printf("小数点文字=[%s]¥n", lc->decimal_point);
    printf("国際通貨記号=[%s]¥n", lc->int_curr_symbol);
    printf("通貨記号=[%s]¥n", lc->currency_symbol);
    printf("桁グループ区切子=[%s]¥n", lc->thousands_sep);
    printf("負の金額値の符号文字=[%s]¥n", lc->negative_sign);
    printf("金額の小数点以下桁数=%d¥n", lc->frac_digits);
    return 0;
}
```

実行結果
```
小数点文字=[.]
国際通貨記号=[JPY]
通貨記号=[¥]
桁グループ区切子=[,]
負の金額値の符号文字=[-]
金額の小数点以下桁数=0
```

localtime

形式
```
#include <time.h>
struct tm *localtime(const time_t *t);
```
戻値　tm構造体型へのポインタ

説明　time()関数で取得した暦時刻tをtm構造体型の現地時間（日本時間）にする。tm構造体については「時間表現用データ型」（p. 229）を参照。

用例　→「現在時刻を表示する」（p. 231）

log

形式
```
#include <math.h>
double log(double x);
```
戻値　計算結果

説明　*e*(=2.7182…)を底とするxの自然対数を返す。

用例
```
d = log(2.718282);    // dは1.0になる
```

log10

形式
```
#include <math.h>
double log10(double x);
```
戻値　計算結果

説明　常用対数（10を底とするxの対数）を返す。

用例
```
d = log10(100.0);     // dは2.0になる
```

longjmp

形式
```
#include <setjmp.h>
void longjmp(jmp_buf env, int val);
```
戻値　なし

説明　setjmp()でセーブされた環境情報であるenvを使って、そのsetjmp()が記述位置に広域ジャンプする。そのときに状態値valをsetjmp()マクロに渡す。この値がsetjmp()マクロに渡され、setjmp()マクロの戻り値となる。その値は0以外でなければならない。もし0を渡すとsetjmp()の戻り値は1に調整される。jmp_buf型はsetjmp.hの中で定義されている。

用例　→setjmp()関数（p. 306）

malloc

形式
```
#include <stdlib.h>
void *malloc(size_t size);
```
戻値　正常時：割り当てたメモリへのポインタ　エラー時：NULL

説明　sizeバイトのメモリ領域を確保し、それへのポインタを返す。確保した領域が初期化されることはない。確保領域は適切に境界調整されている。

用例
```
char *p;
p = (char *)malloc(1000);                  // 1000バイト分メモリを確保する
if (p == NULL) {
    printf("out of memory"); exit(1);      // メモリ確保に失敗した
}
```

mblen

形式 **#include <stdlib.h>**
int mblen(const char *mbs, size_t n);

戻値 多バイト文字のとき：多バイト文字を構成するバイト数
多バイト文字でない：-1
mbsが""のとき：0
mbsがNULLのとき：多バイト文字がシフト状態に依存する表現形式であれば：非0／そうでないなら：0

説明 多バイト文字列mbsの先頭n字までを検査し、多バイト文字を構成するバイト数を返す。通常はnにはMB_CUR_MAXを使う。この関数は正当な多バイト文字であるかどうかの確認に使用できる。

用例 →「157 ワイド文字と多バイト文字の処理1」(p. 234)

mbstowcs

形式 **#include <stdlib.h>**
size_t mbstowcs(wchar_t *wcs, const char *mbs, size_t n);

戻値 変換完了したワイド文字数を返す
無効な多バイト文字が含まれているとき(size_t)-1を返す

説明 多バイト文字列mbsをワイド文字列に変換し、n個以下の文字をwcsに格納する。

用例 →「157 ワイド文字と多バイト文字の処理1」(p. 234)

mbtowc

形式 **#include <stdlib.h>**
int mbtowc(wchar_t *wch, const char *mbs, size_t n);

戻値 多バイト文字を構成するバイト数を返す
正当な多バイト文字でないときは-1を返す
mbsが""のとき：0
mbsがNULLのとき：多バイト文字がシフト状態に依存する表現形式であれば：非0／そうでないなら：0

説明 mbsの最大nバイトを検査し、多バイト文字をワイド文字に変換してwchに格納する。通常はnにマクロMB_CUR_MAX(多バイト文字の最大構成バイト数)を指定する。

用例 →「157 ワイド文字と多バイト文字の処理1」(p. 234)

memchr

形式 **#include <string.h>**
void *memchr(const void *s, int ch, size_t n);

戻値 見つかった文字へのポインタ　見つからないとき：NULL

説明 sの先頭からn文字をサーチし(unsigned charとして判定)文字chのある位置を返す。¥0があっても検索をやめることはない(¥0も検索できる)。memchr(), memcmp(), memcpy(), memmove(), memset()の各関数は指定されたメモリ領域を処理対象にする。処理対象を「文字列」として認識することはないので¥0文字も単なるデータとしてあつかわれる。

用例

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    char *p, s[] = "ABCDEFGHIJ";              // この文字列設定をする
    s[5] = '¥0';                              // 途中文字を0にする
    p = strchr(s, 'H');                       // strchr()では見つからない
    printf("[%s]¥n", p==NULL ? "" : p);       // 出力：[]
    p = memchr(s, 'H', sizeof(s));            // memchr()では見つかる
    printf("[%s]¥n", p==NULL ? "" : p);       // 出力：[HIJ]
    return 0;
}
```

memcmp

形式
```
#include <string.h>
int memcmp(const void *s1, const void *s2, size_t n);
```
戻値 s1>s2なら正の値、s1=s2なら0, s1<s2なら負の値

説明 s1とs2のそれぞれはじめのnバイトのデータを比較する。

用例 int型配列dt1, dt2の5要素分を比較する。
```
int d1[] = {10, 20, 30, 40, 50};
int d2[] = {10, 20, 30, 40, 50};

if (memcmp(d1,d2,sizeof(int)*5) == 0) puts("equal");        // 出力：equal
else                                  puts("not equ");
```

memcpy

形式
```
#include <string.h>
void *memcpy(void *s1, const void *s2, size_t n);
```
戻値 コピー後のs1を返す

説明 s2の最初のnバイトをs1にコピーする。コピー元とコピー先が重なっている場合の動作は不定。このときはmemmove()を使うと正しくコピーされる。memcpy()は文字列コピーではないので整数配列の一括コピーにも利用できる。最後に¥0が自動付加されることもない。

用例
```
int i, d1[5], d2[5] = {10, 11, 12, 13, 14};
memcpy(d1, d2, sizeof(d2));      // int型配列をコピー
for (i=0; i<5; i++)
    printf("%d ", d1[i]);        // 出力：10 11 12 13 14
putchar('¥n');
```

memmove

形式
```
#include <string.h>
void *memmove(void *s1, const void *s2, size_t n);
```
戻値 s1を返す

説明 memcpy()と同じ処理をする(s2の最初のnバイトをs1にコピーする)。memcpy()と異なるのはコピー元とコピー先が重なっている場合にも正しくコピーされることである。

用例
```
char s[80] = "ABCDEFG";
memmove(s+3, s, strlen(s)+1);     // 重複コピーも安全
puts(s);                          // 出力：ABCABCDEFG
```

memset

形式
```
#include <string.h>
void *memset(void *s, int ch, size_t n);
```
戻値 sを返す

説明 sの指す領域の最初のnバイトを文字ch(unsigned charに変換する)で満たす。

用例
```
char s[] = "ABCDEFGH";
memset(s, 'a', 4);      // 先頭4文字を'a'にする
printf("%s¥n", s);      // 出力：aaaaEFGH
```

mktime

形式
```
#include <time.h>
time_t mktime(struct tm *tmptr);
```
戻値 変換された暦時刻　エラー時：(time_t)(-1)

説明 tmptrの指すtm型構造体のデータを暦時刻(time_t型)に変換する。メンバの値は適正な範囲をオーバーしていても合理的に処理される。たとえば分指定が65であっても正しく処理する。

用例　現在の時間から1日と3661秒経過後の時間を知る

```
#include <stdio.h>
#include <time.h>
int main(void)
{
    time_t mytime;
    struct tm *ltime;

    time(&mytime);
    ltime = localtime(&mytime);
    printf("時間1:%s", asctime(ltime));

    ltime->tm_sec +=  3661;          // 3661秒増やす
    ltime->tm_mday +=  1;            // 1日増やす

    mytime = mktime(ltime);          // 一度暦時刻に戻して適性化し
    ltime = localtime(&mytime);      // それを再びtm構造体に変換する
    printf("時間2:%s", asctime(ltime));
    return 0;
}
```

実行結果

```
時間1:Mon Mar 08 11:47:30 2010    1日と1時間と1分と1秒進んでいる
時間2:Tue Mar 09 12:48:31 2010    曜日も自動的に進んでいる
```

modf

形式

```
#include <math.h>
double modf(double x, double *iptr);
```

戻値　計算結果

説明　xを整数部と小数部に分割し、整数部を*iptrに入れ、小数部を返す。

用例

```
double a, b;
b = modf(12.34, &a);   // aは12.0に、bは0.34になる
```

perror

形式

```
#include <stdio.h>
void perror(const char *s);
```

戻値　なし

説明　stderrにsを出力する。次に": "に続いてerrno(システムのエラー番号が格納されている)に対応するエラーメッセージを表示する。ユーザー自身が表示したいメッセージがないときは引数をNULLにする。参考：表示されるシステムエラーメッセージはstrerror(errno)と同じである。

用例

```
if ((fp=fopen("moexist.txt","r")) == NULL) {    // 存在しないファイルをオープン
    perror("不正な処理です"); exit(1);
}
```

実行結果

```
不正な処理です: No such file or directory    ——ある処理系ではこう表示
```

pow

形式

```
#include <math.h>
double pow(double x, double y);
```

戻値　計算結果

説明　xのy乗を計算する。

用例

```
d = pow(2.0, 3.0);    // dは8.0になる
```

printf

形式
```
#include <stdio.h>
int printf(const char *format, ...);
```

戻値 正常時：出力した文字数　エラー時：負の値

説明 対応する引数の内容をformatで指定する書式にしたがいstdoutに出力する。formatの記述方法については「133 書式つき出力2」(p. 186) を参照。

用例
```
printf("%d¥n", idt);        // 変数idtの値を10進数で出力
```

putc

形式
```
#include <stdio.h>
int putc(int ch, FILE *stream);
```

戻値 正常時：出力した文字　エラー時：エラー表示子をセットしEOFを返す

説明 文字chをstreamが指すストリームに出力する。マクロとして実現されることがあること以外はfputc()と等価である。

用例
```
putc('a', fp);        // ファイルfpに文字'a'を書き込む
```

putchar

形式
```
#include <stdio.h>
int putchar(int ch);
```

戻値 正常時：引数c　エラー時：エラー表示子をセットしEOFを返す

説明 文字chをstdoutに出力する。

用例
```
putchar('a');        // 'a'を出力する
```

puts

形式
```
#include <stdio.h>
int puts(char *s);
```

戻値 正常時：非負　エラー時：EOF

説明 文字列sをstdoutへ出力する。最後に改行文字を出力する。

用例
```
puts("Hello");        // "Hello"と出力して改行する
```

qsort

形式
```
#include <stdlib.h>
void qsort(void *base, size_t n, size_t size,
           int (*cmp)(const void *a, const void *b));
```

戻値 なし

説明 クイックソートを行なう。qsort()関数は、配列base[0]～base[n-1]の、1要素sizeバイトの、データをソートする。そのとき要素の大小比較用にユーザー提供関数cmp()(bsearch()用と同じ)を使用する。

用例 int型配列であるdt[0]～dt[9]をソートする
```
#include <stdio.h>
#include <stdlib.h>
int intcmp(const void *a, const void *b)    // int昇順比較関数
{
    return *(int *)a - *(int *)b;
}

int main(void)
{
    int i, dt[10] = {45, 65, 23, 87, 53, 67, 32, 19, 73, 66};
    qsort(dt, 10, sizeof(int), intcmp);
    for (i=0; i<=9; i++)
        printf("%d ", dt[i]);        // 出力：19 23 32 45 53 65 66 67 73 87
    printf("¥n");
```

```
    return 0;
}
```

raise

形式
```
#include <signal.h>
int raise(int sig);
```

戻値 正常時：0　エラー時：非0

説明 sigで指定したシグナルを発生し、sig対応の動作を実行する。sigで指定できる標準のシグナルとしてSIGABRT, SIGFPE, SIGILL, SIGINT, SIGSEGV, SIGTERMがある。→signal()関数(p. 309)

用例
```
#include <stdio.h>
#include <signal.h>
int main(void)
{
    puts("start");
    raise(SIGABRT);  // 強制的に「異常終了」を発生させる
    puts("end");
    return 0;
}
```

実行結果
```
start     ――異常終了のためendは表示されない
```

rand

形式
```
#include <stdlib.h>
int rand(void);
```

戻値 疑似乱数を返す

説明 0～RAND_MAXまでの疑似乱数整数を返す。RAND_MAXはstdlib.hの中で定義されていて、32767(7FFFh)以上と規定されている。

用例
```
n = rand();   // 乱数を返す
```

realloc

形式
```
#include <stdlib.h>
void *realloc(void *ptr, size_t newsize);
```

戻値 正常時：再確保されたメモリ領域へのポインタ　エラー時：NULL

説明 malloc(), calloc(), realloc()関数によって以前に確保されたptrで示されるメモリ領域を、newsizeで示すサイズに再割り当てする。以前に確保されたポインタと一致しないときや、ptrで示す領域がすでに解放済みのときの動作は未定義。古いメモリブロックにあったデータは確保サイズ内で維持される。新領域確保が失敗し、戻り値がNULLの場合も、元のptrで示す内容は維持される。確保領域は適切に境界調整されている。

用例
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void memput(char *p, int n)          // アドレスpからn文字出力
{
    int i;
    for (i=0; i<n; i++) { putchar(*p++); }
    putchar('¥n');
}

int main(void)
{
    char *p1, *p2, *p3;
    p1 = (char *)malloc(100);          // 100バイト確保
    strcpy(p1, "abcdefgh");
```

```
    memput(p1, 8);                          // 出力:abcdefgh
    p2 = (char *)realloc(p1, 200);          // 200バイトに再確保
    memput(p2, 8);                          // 出力:abcdefgh
    p3 = (char *)realloc(p2, 5);            // 5バイトに再確保
    memput(p3, 5);                          // 出力:abcde
    return 0;
}
```

remove

形式
```
#include <stdio.h>
int remove(const char *filename);
```

戻値 正常時:0　エラー時:非0

説明 filenameで示すファイルを削除する。対象ファイルがオープンされているときの動作は処理系定義である。

用例
```
if (remove("tst.txt") == 0)     // tst.txtを削除する
    puts("削除しました");
else
    puts("削除できません");
```

rename

形式
```
#include <stdio.h>
int rename(const char *oldname, const char *newname);
```

戻値 正常時:0　エラー時:非0

説明 ファイル名をoldnameからnewnameに変更する。ファイルnewnameがすでに存在するときの動作は処理系定義である。

用例
```
rename("tst.txt", "new.txt");          // tst.txtをnew.txtにリネームする
```

rewind

形式
```
#include <stdio.h>
void rewind(FILE *stream);
```

戻値 なし

説明 streamが指すストリームのファイル位置表示子を先頭(0L)にしエラー指示子をクリアする。ファイル位置表示子移動に関してはfseek(fp, 0L, SEEK_SET);と同じ。

用例
```
rewind(fp);          // ファイルfpを先頭から読み書きできるようにする
```

scanf

形式
```
#include <stdio.h>
int scanf(const char *format, ...);
```

戻値 正常時:入力データの個数　入力がひとつもなくエラー時:EOF

説明 stdinから書式formatにしたがい対応する引数にデータを読み込む。formatの記述方法については「136 書式つき入力2」(p. 192)を参照。

用例
```
int idt;
scanf("%d", &idt);   // int型数値入力
```

setbuf

形式
```
#include <stdio.h>
void setbuf(FILE *stream, char *buf);
```

戻値 なし

説明 streamが指すストリームの入出力バッファをbufに指定する。用意するbufのサイズはBUFSIZで確保する。BUFSIZはstdio.hの中で定義されている適切な値である。bufの代わりにNULLを使うとバッファリングなしを指定する。setbuf()関数はsetvbuf()関数(p. 307)の簡略形である。

用例 次のsetvbuf()とsetbuf()の記述は同じ設定を行なう。ただしsetbuf()は戻り値がない点が異なる

```
char buf[BUFSIZ];
setvbuf(fp, buf, _IOFBF, BUFSIZ);    // 完全バッファリングにする
setbuf(fp, buf);                     // 同上

setvbuf(fp, NULL, _IONBF, 0);        // 非バッファリングにする
setbuf(fp, NULL);                    // 同上
```

setjmp

形式 **#include <setjmp.h>**
int setjmp(jmp_buf env);

戻値 直接呼び出しのとき：0
longjmp()からの呼び出しのとき：longjmp()のリターンコード
longjmp()のリターンコードが0のとき：1

説明 この位置に戻るための環境情報をenvに設定し、longjmp()にそなえるためのマクロである。longjmp()関数が実行されるとこの位置に広域ジャンプしてくる。複数のjmp_buf型変数を用意すれば、複数の広域ジャンプルートを確保できる。

```
jmp_buf jpos1, jpos2, jpos3;                  ── 3つの変数を宣言
setjmp(jpos1)  ↔  longjmp(jpos1, 1) ┐
setjmp(jpos2)  ↔  longjmp(jpos2, 1) ├3種のジャンプが可。
setjmp(jpos3)  ↔  longjmp(jpos3, 1) ┘
```

用例 (3)で設定し(4)→(1)→(2)→(3)とジャンプする

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>                    /* for isalpha() */
#include <setjmp.h>                   /* for setjmp longjmp() */
jmp_buf jpos;                         // 環境情報用変数
void func(void)                       // (1)
{
    puts("func:start");
    puts("longjmpでジャンプします.");
    longjmp(jpos, 1);                 // (2) jposの内容でジャンプ
    puts("func:end");
}

int main(void)
{
    int ret;
    puts("main:start");
    ret = setjmp(jpos);               // (3) 戻り環境設定。またlongjmpからの入口
    switch (ret) {
    case 0:                           // 最初の処理のとき
        puts("現在の環境をjposにセーブしました。"); break;
    case 1:                           // longjmpで戻ってきたとき
        puts("longjmpでジャンプしてきました。"); exit(1);
    }
    func();                           // (4)
    puts("main:end");
    return 0;
}
```

実行結果

```
main:start
現在の環境をjposにセーブしました。
func:start
longjmpでジャンプします.
longjmpでジャンプしてきました。
```

setlocale

形式 `#include <locale.h>`

`char *setlocale(int category, const char *locale);`

戻値 正常時：設定したロケールを示す文字列へのポインタ　エラー時：NULL

localeがNULLの場合は現在のロケールを示す文字列へのポインタ

説明 ロケールを設定する。設定内容はlconv構造体に格納される（localeconv()関数（p. 298））。ロケールとは国や文化や言語に依存する約束ごと全般のことで、地域規約あるいは地域仕様ともいう。通貨記号（¥や$）や、日付/時刻の表現方法などがこれに属する。categoryでカテゴリ（設定や検索したい部分）を指定し、localeで設定したい地域情報を指定する。設定内容は文字操作や比較関数（例：strcoll()）などに国状を反映させるようになる。引数localeとして"C"を指定するとCプログラム実行のための最小環境を設定する。引数localeとして""を定義すると、処理系定義のデフォルトの地域性をもつ設定になる。その他のlocale指定可能項目は処理系依存である。たとえばVisual C++では"jpn"や"usa"を指定できる。プログラム起動時はsetlocale(LC_ALL, "C");の設定になっている。categoryとして指定できる項目は次のとおりである。

category指定

名前	影響する動作（関数例）
LC_ALL	ロケール全体
LC_COLLATE	文字列比較動作。strcoll(), strxfrm()関数
LC_CTYPE	文字操作関数（isxxx()関数）、多バイト文字・ワイド文字関連関数
LC_MONETARY	localeconv()関数の返す金額書式化情報
LC_NUMERIC	localeconv()関数の返す非金額書式化情報 書式付入出力関数や文字列変換関数の小数点文字
LC_TIME	日付・時刻表示strftime()関数

用例1 Visual C++の例

```
setlocale(LC_ALL, "C");              // "C"ロケールを設定する
setlocale(LC_ALL, "");               // デフォルトの地域性をもつ設定
setlocale(LC_ALL, "jpn");            // Visual C++ ではこの記述が有効
setlocale(LC_ALL, "usa");            // Visual C++ ではこの記述が有効
setlocale(LC_MONETARY, "usa");       // 通貨形式だけ変更する
```

用例2 →localeconv()関数（p. 298）

setvbuf

形式 `#include <stdio.h>`

`int setvbuf(FILE *stream, char *buf, int mode, size_t size);`

戻値 正常時：0　エラー時：非0

説明 streamが指すストリームの入出力バッファをbufに指定する。用意するbufのサイズをsizeで指定する。modeは次のものを設定できる。

_IOFBF	完全バッファリング
_IOLBF	行バッファリング
_IONBF	非バッファリング

setvbuf()はデフォルトのバッファでは不満な場合に使用するものである。完全バッファリングはバッファ領域が満杯になったとき、入出力転送動作を行なう。行バッファリングはバッファ満杯時および改行文字がきたときに、入出力転送動作を行なう。非バッファリングは1バイトごとに実際の入出力転送動作を行なうモードである。setvbuf()はストリームをオープンした直後に実行させる。適切なバッファサイズを示すBUFSIZがstdio.hの中で定義されているのでこれを使うとよい。bufとしてNULLを指定すると自動的にsizeバイトの領域を割り付け、

それをバッファとして使う。setvbuf()はsetbuf()関数(p. 305)と似ているが、バッファのモードとサイズを指定でき、戻り値をもつ。

用例 通常Windows環境のstdoutは非バッファリングであり、setvbuf(fp, NULL, _IONBF, 0);が実行されていると考えることができる。これを完全バッファリングありに変更する。なおGNU Cでstdoutはデフォルトで行バッファリングになっている。

```
#include <stdio.h>
#include <time.h>                         /* for clock() */
void mywait(int msec)                     // 1ミリ秒単位で指定した時間だけ待ち合わせる
{
    clock_t start = clock();
    while ((clock() - start) / (CLOCKS_PER_SEC/1000.0) < msec)
        ;
    return;
}

int main(void)
{
    int i;
    char mybuf[BUFSIZ];                   // 独自に用意したバッファ領域
  // setvbuf(stdout, NULL, _IONBF, 0);
                                   // 注1:GNU C(Linux)の場合はこれを入れると確認できる

    printf("BUFSIZ=%d¥n", BUFSIZ);        // 参考のためにBUFSIZを表示
    for (i='a'; i<='j'; i++) {            // バッファリングなしなので
        putchar(i);                       // 時間待ちしながらトコトコと表示
        mywait(200);                      // 200ミリ秒間待つ
    }

    setvbuf(stdout, mybuf, _IOFBF, BUFSIZ); // mybufで完全バッファリングすると
    for (i='A'; i<='J'; i++) {
        putchar(i);                       // バッファに貯められて
        mywait(200);
    }
    fflush(stdout);                       // このフラッシュ指示で一気に表示される
    return 0;
}
```

実行結果1:Visual C++で実行

```
BUFSIZ=512                 ――サイズは512
abcdefghijABCDEFGHIJ       ――a~jはトコトコと、A~Jは一気に表示される
```

実行結果2:GNU Cで実行(注1のコメントを外す)

```
BUFSIZ=8192                ――サイズは8192である
abcdefghijABCDEFGHIJ
```

注:CLOCKS_PER_SEC定数の値は整数であり、100という値をもつものがある。その場合100/1000は0になるので、CLOCKS_PER_SEC/1000.0にしている。

note Win32環境での行バッファ

Visual C++で行バッファリングをするつもりで、

```
setvbuf(stdout, mybuf, _IOLBF, BUFSIZ);
```

を記述しても完全バッファリングになってしまう。Visual C++で**setvbuf()**関数のヘルプを見ると「Win32環境の場合、**IOLBF**は**IOFBF**と同じである」という旨の明記がある。

signal

形式
```
#include <signal.h>
void (*signal(int sig, void (*func)(int)))(int);
```

戻値 シグナルハンドラへのポインタfunc　エラー時：SIG_ERR(正の値をerrnoに格納)

説明 シグナル処理とは外部機器からの割込みやプログラム中での非同期割込みなどを処理するものである。signal()関数はシグナルが検出されたときの対処方法をあらかじめ設定しておく役目をもつ。シグナル処理の種類はsigで指定する。sigシグナルはraise()関数(p. 304)によって発生させられる。各シグナルに対して適切な処理をする関数が必要でこれをシグナルハンドラといい、funcで指定する。funcとしてSIG_IGNを指定するとそのシグナルは無視(IGNore)する。funcとしてSIG_DFLを指定すると標準の処理(DeFauLt)に戻す。それ以外では指定されたハンドラ関数が実行される。定義済みのシグナル定数として次のものがある。

標準シグナル定数

SIGABRT	異常終了
SIGFPE	算術エラー
SIGILL	不正命令
SIGINT	割込み(WindowsではCTRL+C)
SIGSEGV	メモリへの不正アクセス
SIGTERM	終了要求

注：signal()関数には処理系定義要素が多いので注意して使用する必要がある。

用例 異常終了が発生したときに、独自の情報を表示する。abort()で意図的に異常終了を発生させる

```
#include <stdio.h>
#include <stdlib.h>   /* for abort()  */
#include <signal.h>   /* for signal() */
int mystatus;                          // 状態記憶用変数
void myhandle(int sig)                 // sigにはシグナル番号が渡される
{
    printf("シグナル%d番が起動されました。¥n", sig);
    printf("status=%d でプログラムを中止しました。¥n", mystatus);
    printf("問い合わせの場合はこの番号をお伝えください。¥n");
}

int main(void)
{
    signal(SIGABRT, myhandle);         // 異常終了のとき関数myhandleを呼ぶ
    mystatus = 1234;
    puts("----abort前");
    abort();                           // わざと異常終了発生
    puts("----abort後");
    return 0;
}
```

実行結果1：ある処理系ではこのように表示

```
----abort前
シグナル6番が起動されました。
status=1234 でプログラムを中止しました。
問い合わせの場合はこの番号をお伝えください。
アボートしました (コアダンプ)
```

実行結果2：別の処理系ではこのように表示

```
----abort前

This application has requested (途中省略) for more information.
シグナル22番が起動されました。
status=1234 でプログラムを中止しました。
問い合わせの場合はこの番号をお伝えください。
```

note **signal関数の読み方**

signal() 関数のプロトタイプは非常に面倒な宣言になっている。参考のために読み方を示す。

```
void (*signal(int sig, void (*func)(int)))(int);
```

これは関数である **signal()** を宣言する。**signal()** 関数は **void** を返す関数へのポインタを返す。

signal() 関数はふたつの引数をもつ。ひとつ目の引数は **int** 型の **sig** である。ふたつ目の引数 **func** は「**int** 型の引数をもち **void** 型である関数へのポインタ」である。

signal() が返すポインタが指す関数は、ひとつの **int** 型の引数をもつ。

sin

形式
```
#include <math.h>
double sin(double x);
```
戻値 計算結果

説明 サイン値（正弦）を返す。

用例
```
d = sin(30*3.14159/180.0);          // dは0.500000。サイン30度
```

sinh

形式
```
#include <math.h>
double sinh(double x);
```
戻値 計算結果

説明 ハイパボリックサイン値（双曲線正弦）を返す。

sprintf

形式
```
#include <stdio.h>
int sprintf(char *s, const char *format, ...);
```
戻値 正常時：sに書き込んだ文字数（終端の'¥0'文字は数えない）　エラー時：負の値

説明 対応する引数の内容をformatで指定する書式にしたがい文字配列sに出力する。formatの記述方法については「133 書式つき出力2」（p. 186）を参照。

用例
```
char st[80];
sprintf(st, "%c%d", 65, 123);     // stは A123
```

sqrt

形式
```
#include <math.h>
double sqrt(double x);
```
戻値 計算結果

説明 xの平方根を計算する。

用例
```
d = sqrt(2.0);            // dは1.414214になる
```

srand

形式
```
#include <stdlib.h>
void srand(unsigned int seed);
```
戻値 なし

説明 疑似乱数の発生系列をseed値によって変更する。プログラム起動時はsrand(1)が実行されたのと同じ設定になる。

用例
```
srand(5);                             // 新しい乱数系列にする
srand((unsigned)time(NULL));          // 起動時に異なる乱数を発生させる定番用法
```

sscanf

形式
```
#include <stdio.h>
int sscanf(const char *s, const char *format, ...);
```

戻値 正常時：入力データの個数　エラー時：EOF

説明 文字列sから、書式formatにしたがって、対応する引数にデータを読み込む。formatの記述方法については「136 書式つき入力2」(p. 192) を参照。

用例
```
char st[80] = "B456";
int ch, dt;
sscanf(st, "%c%d", &ch, &dt);          // 1文字をchに数値をdtに読み込む
printf("ch=%c dt=%d¥n", ch, dt);       // 出力：ch=B dt=456
```

strcat

形式
```
#include <string.h>
char *strcat(char *s1, const char *s2);
```

戻値 s1を返す

説明 文字列s1の後ろに文字列s2を連結する。

用例
```
char s[10] = "ABC";
strcat(s, "xyz");        // sは："ABCxyz"
```

strchr

形式
```
#include <string.h>
char *strchr(const char *s, int ch);
```

戻値 見つかったとき：その文字へのポインタ　見つからないとき：NULL

説明 文字列sの先頭から文字chを探しその位置を返す。chとして'¥0'も指定できる。

用例
```
char *p;
char s[] = "strchr";
p = strchr(s, 'r');
printf("%s¥n", p);         // 出力：rchr
p = strchr(s, '¥0');       // pは末尾の'¥0'位置を指す
```

strcoll

形式
```
#include <string.h>
int strcoll(const char *s1, const char *s2);
```

戻値 s1>s2なら正の値、s1=s2なら0, s1<s2なら負の値

説明 文字列s1と文字列s2の内容を現在の地域仕様にしたがって比較する。その国の文字セットがASCII順番で不都合のあるときに用いる。不都合のないときはstrcmp()関数を使えばよい。strcollはSTRing COLLate(文字列照合)である。

用例
```
if (strcoll(s1, s2) == 0) {        // 文字列s1とs2が等しいなら
    (何かの処理)
}
```

strcmp

形式
```
#include <string.h>
int strcmp(const char *s1, const char *s2);
```

戻値 s1>s2なら正の値、s1=s2なら0, s1<s2なら負の値

説明 文字列s1と文字列s2の内容を辞書順に比較する。

用例
```
if (strcmp(s, "str") == 0) {       // 文字列sが"str"なら
    (何かの処理)
}
```

strcpy

形式
```
#include <string.h>
char *strcpy(char *s1, const char *s2);
```
戻値 s1を返す

説明 文字配列s1に文字列s2をコピーする。

用例
```
strcpy(s, "Hello");   // sは："Hello"
```

strcspn

形式
```
#include <string.h>
size_t strcspn(const char *s1, const char *s2);
```
戻値 先頭からの文字数

説明 文字列s1のうち、文字列s2にふくまれるどれかの文字が最初に現れる直前までの文字数を返す。いいかえると文字列s2にふくまれない文字だけで構成される文字列s1の先頭からの連続文字数を返す。

用例
```
n = strcspn("137xyz3a567","abcde");      // nは7
```

strerror

形式
```
#include <string.h>
char *strerror(int errnum);
```
戻値 システムのもっているエラーメッセージへのポインタ

説明 errnumに対応した処理系定義のエラーメッセージへのポインタを返す。メッセージの内容は処理系により異なる。必要なヘッダファイル名がstring.hなので注意。

用例
```
for (i=0; i<=5; i++)
    printf("%d:%s¥n", i, strerror(i));
```

実行結果：ある処理系の場合
```
0:No error
1:Operation not permitted
2:No such file or directory
3:No such process
4:Interrupted function call
5:Input/output error
```
C処理系が内部にもっている0〜5番のエラーメッセージ

strftime

形式
```
#include <time.h>
size_t strftime(char *s, size_t maxsize, const char *format,
                                const struct tm *tmptr);
```
戻値 正常時：結果の文字数を返す エラー時：0

説明 時刻と日付を地域の表示形式にする。tmptrが指すtm構造体内容をformatで示した内容にしたがって変換し、maxsizeで示す文字数(¥0を含む)以内でsに格納する。変換指定は%ではじまる指定文字で行なう。変換指定以外の文字はそのまま表示される。主要な変換指定文字にはC89規格で%a %A %b %B %c %d %H %I %j %m %M %p %S %U %w %W %x %X %y %Y %Z %%がある。サンプルプログラムで用例を示す。

C99ではさらに次の変換指定文字が追加されている。

文字	説明
%C	西暦年の上2桁
%D	%m/%d/%yと等価な日付表現
%e	日(1〜31)。1けたの数のときは空白文字を前置する
%F	%Y-%m-%dと同じ
%g	暦週にもとづく4桁の西暦年の下2桁
%G	暦週にもとづく4桁の西暦年
%h	%bと同じ
%n	改行
%r	ロケール依存の12時間制での時刻表現(午前/午後形式)
%R	24時間表記での時刻。%H:%Mと同じ

%t	タブ文字
%T	%H:%M:%Sと同じ
%u	曜日を表現する値(1～7)。月曜日が1。JIS X 0301の規定による。
%V	年始からの週番号(01～53)。JIS X 0301の規定による。
%z	UTCからの時差

注：変換文字の前にEまたはOを置いた代替変換文字もあるが、説明は略する。

用例

```
#include <stdio.h>
#include <time.h>
int main(void)
{
    time_t mytime;
    struct tm *ltm;
    char s[80];

    time(&mytime);                                          // 暦時刻取得
    ltm = localtime(&mytime);                               // 現地時間に変換
    printf("%s", asctime(ltm));                             // 今日の日付時刻を表示

    strftime(s, 80, "a[%a]", ltm); puts(s);                 // 曜日略称
    strftime(s, 80, "A[%A]", ltm); puts(s);                 // 曜日
    strftime(s, 80, "b[%b]", ltm); puts(s);                 // 月略称
    strftime(s, 80, "B[%B]", ltm); puts(s);                 // 月
    strftime(s, 80, "c[%c]", ltm); puts(s);                 // 地域仕様日付時刻
    strftime(s, 80, "d[%d]", ltm); puts(s);                 // 日(1～31)
    strftime(s, 80, "H[%H]", ltm); puts(s);                 // 24時間表記の時間(0～23)
    strftime(s, 80, "I[%I]", ltm); puts(s);                 // 12時間表記の時間(1～12)
    strftime(s, 80, "j[%j]", ltm); puts(s);                 // 1月1日からの日数(1～366)
    strftime(s, 80, "m[%m]", ltm); puts(s);                 // 月(1～12)
    strftime(s, 80, "M[%M]", ltm); puts(s);                 // 分(0～59)
    strftime(s, 80, "p[%p]", ltm); puts(s);                 // 午前午後
    strftime(s, 80, "S[%S]", ltm); puts(s);                 // 秒(0～59)
    strftime(s, 80, "U[%U]", ltm); puts(s);                 // 通算週(日曜基準, 0～53)
    strftime(s, 80, "w[%w]", ltm); puts(s);                 // 曜日(日曜が0)
    strftime(s, 80, "W[%W]", ltm); puts(s);                 // 通算週(月曜基準, 0～53)
    strftime(s, 80, "x[%x]", ltm); puts(s);                 // 地域表現日付
    strftime(s, 80, "X[%X]", ltm); puts(s);                 // 地域表現時刻
    strftime(s, 80, "y[%y]", ltm); puts(s);                 // 下2桁西暦(00～99)
    strftime(s, 80, "Y[%Y]", ltm); puts(s);                 // 西暦4桁
    strftime(s, 80, "Z[%Z]", ltm); puts(s);                 // 時間帯名(不明のときはなし)
    strftime(s, 80, "[%%]",  ltm); puts(s);                 // %自身
    return 0;
}
```

実行結果：Visual C++で実行

```
Mon Mar 08 12:54:38 2010
a[Mon]
A[Monday]
b[Mar]
B[March]
c[03/08/10 12:54:38]
d[08]
H[12]
I[12]
j[067]
m[03]
M[54]
p[PM]
S[38]
U[10]
w[1]
W[10]
x[03/08/10]
X[12:54:38]
y[10]
Y[2010]
Z[東京 (標準時)]
[%]
```

strlen

形式

```
#include <string.h>
size_t strlen(const char *s);
```

戻値 文字列の長さを返す

説明 文字列sの長さを返す。終端文字'¥0'は数えない。

用例

```
n = strlen("abcde");     // nは5
```

strncat

形式

```
#include <string.h>
char *strncat(char *s1, const char *s2, size_t n);
```

戻値 s1を返す

説明 文字配列s1の後ろに文字列s2の先頭からn個までの文字を連結する。文字列の最後に¥0を付加する。

用例

```
char s1[10] = "abcde", s2[10] = "ABCDE";
strncat(s1, s2, 3);      // s1は abcdeABC
```

strncmp

形式

```
#include <string.h>
int strncmp(const char *s1, const char *s2, size_t n);
```

戻値 s1>s2なら正の値、s1=s2なら0, s1<s2なら負の値

説明 文字列s1と文字列s2の先頭n文字同士を辞書順に比較する。

用例

```
if (strncmp(st, "main", 4) == 0) {      // stの先頭4文字が"main"なら
    (何かの処理)
}
```

strncpy

形式

```
#include <string.h>
char *strncpy(char *s1, const char *s2, size_t n);
```

戻値 s1を返す

説明 文字配列s1に文字列s2の先頭のn個までの文字をコピーする。'¥0'の自動付加は行なわれないので注意。必要なら自分で入れる。ただしs2の長さがnより小さいときはnに達するまでs1に'¥0'を埋める。

用例

```
int i;
char s1[10] = "ABCDEFGH", s2[10] = "1234";
strncpy(s1, s2, 3);          // s2の先頭3文字だけコピー
puts(s1);                    // 出力：123DEFGH
strncpy(s1, s2, 6);          // 6より少ない分だけ'¥0'を埋める
for(i=0; i<8; i++)
    printf("%X ", s1[i]);    // 出力：31 32 33 34 0 0 47 48
```

strpbrk

形式

```
#include <string.h>
char *strpbrk(const char *s1, const char *s2);
```

戻値 見つかった文字へのポインタ　見つからないとき：NULL

説明 文字列s1の中で、文字列s2にふくまれるどれかの文字が現れる位置のポインタを返す。strchr()と似ているが、strchr()は単一文字を検索し、strpbrk()は複数の検索候補文字を指定できることが異なる。

用例

```
char *p, s[] = "void func(char s*)";
p = strpbrk(s, "()[]{}");
printf("%s¥n", p);          // 出力：(char s*)
```

strrchr

形式

```
#include <string.h>
char *strrchr(const char *s, int ch);
```

戻値 見つかったとき：その文字へのポインタ　見つからないとき：NULL

説明 文字列sの後ろから文字chを探しその位置を返す。chとして'¥0'も指定できる。

用例

```
char *p;
char s[] = "strrchr-test";
p = strrchr(s, 'r');
printf("%s¥n", p);          // 出力：r-test
p = strrchr(s, '¥0');       // pは末尾の'¥0'位置を指す
```

strspn

形式

```
#include <string.h>
size_t strspn(const char *s1, const char *s2);
```

戻値 先頭からの文字数

説明 文字列s1の中で文字列s2にふくまれない文字が最初に現れる直前までの文字数を返す。いいかえると文字列s2にふくまれる文字だけで構成される、文字列s1の先頭からの連続文字数を返す。spnはspan(間隔)のこと。

用例

```
n = strspn("cdbxde", "abcde");    // nは3
```

strstr

形式

```
#include <string.h>
char *strstr(const char *s1, const char *s2);
```

戻値 見つかった文字列へのポインタ　s2が""のとき：s1　見つからないとき：NULL

説明 文字列s1の先頭から文字列s2を探し、見つかった位置のポインタを返す。strchr()との違いは第2引数が文字列であること。

用例

```
char *p, s[] = "strstr-test";
p = strstr(s, "tes");
printf("%s¥n", p);          // 出力：test
```

strtod

形式
```
#include <stdlib.h>
double strtod(const char *s, char **endptr);
```

戻値 正常時：変換されたdouble値
答が表現可能範囲外：結果の符号により正または負のHUGE_VALを返し、errnoにエラー値ERANGEを格納
答がアンダーフロー：最小の正の数以下の絶対値（通常0.0）
変換が不可能：0.0
HUGE_VALはmath.hで、ERANGEはerrno.hで定義されている

説明 文字列sの最初の部分をdouble型の値に変換する。変換されなかった末尾側の文字列のポインタがendptrに入る。endptrがNULLのときはこの末尾文字列の処理は行なわれない。〈変換指定〉には先行空白、+-符号、eE指数を含めることができる。C99では0x, 0Xではじまる16進実数値も使うことができる。

用例
```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    char *ep;
    double d;
    d = strtod("3456.789abc", &ep);
    printf("d=%f ep=%s¥n", d, ep);          // 出力：d=3456.789000 ep=abc
    d = strtod("    1.234E3FGH", &ep);
    printf("d=%f ep=%s¥n", d, ep);          // 出力：d=1234.000000 ep=FGH
    return 0;
}
```

strtok

形式
```
#include <string.h>
char *strtok(char *s1, const char *s2);
```

戻値 見つかったトークンへのポインタ　見つからないとき：NULL

説明 文字列s2に含まれる文字を分離記号として、文字列s1の中からトークンを切り出す。最初の呼び出しでは文字列をs1で指定して実行する。この文字列はstrtok()関数内部に保存され最初のトークンへのポインタが返される。2回目以降はs1にNULLを指定して実行する。戻り値がNULLになるまでstrtok()を実行するとすべてのトークンを得ることができる。s2の内容は呼び出すたびに異なっていてもよい。

用例
```
char *p, s[] = "123  ,  456,789   ABC";
p = strtok(s, ", ");                          // ,と空白が分離記号
printf("[%s]¥n", p);                          // 最初のトークン
while ((p=strtok(NULL,", ")) != NULL)         // 2回目以降はNULL指定
    printf("[%s]¥n", p);                      // 2回目以降のトークン
```

実行結果
```
[123]
[456]      得られたトークン
[789]
[ABC]
```

strtol

形式
```
#include <stdlib.h>
long int strtol(const char *s, char **endptr, int base);
```

戻値 正常時：変換されたlong値
答が表現可能範囲外：結果の符号によりLONG_MAXまたはLONG_MINを返し、errnoにエラー値ERANGEを格納
変換が不可能：0
LONG_MAXとLONG_MINはlimits.hで、ERANGEはerrno.hで定義されている

説明 文字列sの最初の部分をlong型の値に変換する。変換されなかった末尾側の文字列のポインタがendptrに入る。endptrがNULLのときはこの末尾文字列の処理は行なわれない。〈変換指定〉には先行空白、+-符号、8進数を示す

0, 16進数を示す0x, 0Xを含めることができる。baseが2～36の値をとるとき、それぞれ基数となる(16なら16進数になる)。a～z(またはA～Z)の文字は10～35の数値に対応する。baseが0のときはCの定数表現にしたがう。0がつけば8進数、0xか0Xがつけば16進数、それ以外は10進数。baseが16のとき0x, 0X表現を用いてもよい。

用例

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
/*#include <limits.h>*/                // LONG_MAXおよびLONG_MINを使うときは指定
int main(void)
{
    char ss[80];
    char *ep;
    long dt;

    strcpy(ss, "100XYZ");                            // "100"までが変換対象になる
    dt = strtol(ss, &ep, 10);                        // 10進数変換
    printf("10:%ld ep=%s¥n", dt, ep);                // 出力:10:100 ep=XYZ
    dt = strtol(ss, &ep, 2);                         // 2進数変換
    printf(" 2:%ld ep=%s¥n", dt, ep);                // 出力: 2:4 ep=XYZ
    dt = strtol(ss, &ep, 8);                         // 8進数変換
    printf(" 8:%ld ep=%s¥n", dt, ep);                // 出力: 8:64 ep=XYZ
    dt = strtol(ss, &ep, 16);                        // 16進数変換
    printf("16:%ld ep=%s¥n", dt, ep);                // 出力:16:256 ep=XYZ
    dt = strtol("0x100XYZ", &ep, 16);                // 先行0xも可
    printf("0x:%ld ep=%s¥n", dt, ep);                // 出力:0x:256 ep=XYZ

    // 以下ep=NULLを指定、base=0 でC定数形式を指定
    dt = strtol("200xyz", (char **)NULL, 0);         // 10進数
    printf("10:%ld¥n", dt);                          // 出力:10:200
    dt = strtol("0200xyz", (char **)NULL, 0);        // 8進数
    printf(" 8:%ld¥n", dt);                          // 出力: 8:128
    dt = strtol("0x200xyz", (char **)NULL, 0);       // 16進数
    printf("16:%ld¥n", dt);                          // 出力:16:512
    return 0;
}
```

注:(char **)NULLは実際にはNULLでもエラーにならない。

strtoul

形式 **#include <stdlib.h>**

unsigned long int strtoul(const char *s, char **endptr, int base);

戻値 正常時:変換されたunsigned longの値

答が表現可能範囲外:ULONG_MAXを返し、errnoにエラー値ERANGEを格納

変換が不可能:0

ULONG_MAXはlimits.hで、ERANGEはerrno.hで定義されている

説明 文字列sをunsigned long型の値に変換する。その他の変換規則はstrtol()と同じ。→strtol()関数(p. 316)

用例

```
char *endp, s[] = "123456789abc";
unsigned long d;
d = strtoul(s, &endp, 10);                      // sを10進数として変換
printf("d=%lu endp=%s¥n", d, endp);             // 出力:d=123456789 endp=abc
```

strxfrm

形式
```
#include <string.h>
size_t strxfrm(char *s1, const char *s2, size_t n);
```

戻値 変換された文字数（終端'¥0'は含めない）

説明 文字列s2を地域仕様にもとづいて変換する。変換後の文字列の先頭から（末尾の'¥0'を含めて）最大nバイトをs1に入れる。その国の文字セットがASCIIコード順と辞書順で不都合のあるときがある。たとえばある国では文字コードは'e','ē'の順だが、辞書順は'ē','e'となる。このような場合は文字列比較にstrcmp()ではなくstrcoll()を使わなくてはならない。しかしstrxfrm()で文字列変形すればstrcmp()を使用できる。日本ではこの関数を使う必要はない。strxfrmはSTRing X(cross) FoRMからの命名である。変換された文字数だけを知りたいときは、ct=strxfrm(NULL, s2, 0);にすればよい。

用例 文字順列に問題があるときでも次のふたつの結果は同じになる

```
char s1[80], s2[80], s1b[80], s2b[80];
(1)  if (strcoll(s1, s2) == 0) { ～ }
(2)  strxform(s1b, s1, 80);
     strxform(s2b, s2, 80);
     if (strcmp(s1b, s2b) == 0) { ～ }
```

system

形式
```
#include <stdlib.h>
int system(const char *cmnd);
```

戻値 cmndがNULLのときの戻り値：　コマンドプロセッサが存在するとき：非0　存在しないとき：0
cmndが文字列のときの戻り値は処理系に依存する（通常、コマンドの返す値）

説明 コマンドプロセッサを呼び出し、cmndで指定したコマンドを実行する。system(NULL);と書いたときはコマンドプロセッサの存在をチェックする。

用例
```
system("type myfile.txt");        // ファイル内容を表示する(DOS窓の場合)
system("cat myfile.txt");         // ファイル内容を表示する(UNIXの場合)
```

tan

形式
```
#include <math.h>
double tan(double x);
```

戻値 計算結果

説明 タンジェント値（正接）を返す。

用例
```
d = tan(30*3.14159/180.0);        // dは0.577350。タンジェント30度
```

tanh

形式
```
#include <math.h>
double tanh(double x);
```

戻値 計算結果

説明 ハイパボリックタンジェント値（双曲線正接）を返す。

time

形式
```
#include <time.h>
time_t time(time_t *t);
```

戻値 現在の暦時刻　エラー時：(time_t)(-1)　　――time_t型にキャストした-1

説明 現在の暦時刻を*tに取得する。*tに入る値と、この関数の戻り値は同じものなので、tdt = time(NULL);のように使ってもよい。このデータはctime()関数(p. 289)を使うと目で読める形になる。

用例 →「現在時刻を表示する」(p. 231)

tmpfile

形式
```
#include <stdio.h>
FILE *tmpfile(void);
```

戻値 正常時：ストリームへのポインタ　エラー時：NULL

説明 一時的に使用するためのファイルをwb+(バイナリ更新モード)でオープンする。このファイルはファイルクローズかプログラム終了で自動的に削除される。

用例
```
#include <stdio.h>
int main(void)
{
    FILE *tmp;
    char s[256];

    if ((tmp=tmpfile()) == NULL) {                // 一時ファイルオープン
        puts("一時ファイルを生成できません。"); return 1;
    }

    fputs("temporary file test", tmp);            // 書き込み
    rewind(tmp);                                  // 先頭に戻して
    fgets(s, 256, tmp);                           // 1行読み込む
    puts(s);
    return 0;
}
```

実行結果：GNU Cで実行した場合
```
temporary file test
```

注：Visual C++ではtmpfile()はルートディレクトリにファイルを生成する。Windows Vistaではルートディレクトリ上のファイル生成を制限しているので管理者権限が必要。

tmpnam

形式
```
#include <stdio.h>
char *tmpnam(char *name);
```

戻値 nameがNULLのときは内部的に生成した名前へのポインタ
nameがNULLでないときはポインタname
適当な文字列が生成できない場合はNULLを返す

説明 一時的にデータを置くためのファイルの名前を作りnameに入れる。この名前はディスク上の既存ファイル名と衝突しないことが保証される。nameがNULLのときは生成した名前を内部的なメモリ領域に置きそのポインタを返す。nameは少なくともサイズがL_tmpnam個であるchar[]型を指している必要がある。この値はstdio.hの中で定義されている。

用例
```
#include <stdio.h>
int main(void)
{
    char name1[L_tmpnam], name2[L_tmpnam], name3[L_tmpnam];
    tmpnam(name1);      // 一時ファイル名を3つ作る
    tmpnam(name2);
    tmpnam(name3);
    puts(name1);        // それを表示する
    puts(name2);
    puts(name3);
    return 0;
}
```

実行結果1：Visual C++の例

```
¥s48.
¥s48.1
¥s48.2
```

実行結果2：GNU Cの例

```
/tmp/filerZPByq
/tmp/filenKTnbK
/tmp/file7Mba03
```

注：同じC処理系でも実行するたびに生成されるファイル名は異なる。

note mkstemp関数

マルチタスク環境では**tmpnam()**関数でファイル名を取得してから、実際にそのファイルをオープンする間に、他のタスクが同一ファイル名でファイルオープンする可能性があるので、UNIX環境では警告が出される。これに対応するものとして、GNU Cではより安全な**mkstemp()**関数の使用を推奨している。またVisual C++でもセキュリティを強化したバージョンとして**tmpnam_s()**関数が用意されている。

tolower

形式 **#include <ctype.h>**
int tolower(int ch);

戻値 対応小文字があるとき：小文字　それ以外のとき：そのままの値

説明 文字chが英大文字なら小文字に変換して返す。

用例
```
c = tolower(c);   // 文字cを小文字にする
```

toupper

形式 **#include <ctype.h>**
int toupper(int ch);

戻値 対応大文字があるとき：大文字　それ以外のとき：そのままの値

説明 文字chが英小文字なら大文字に変換して返す。

用例
```
c = toupper(c);   // 文字cを大文字にする
```

ungetc

形式 **#include <stdio.h>**
int ungetc(int ch, FILE *stream);

戻値 正常時：文字ch　エラー時：EOF

説明 文字chをunsigned char型に変換してstreamが指すストリームに戻す。戻された文字はまだ読んでいなかったものとしてあつかわれる。ungetc()は先読み処理した文字を元の状態に戻すのに使われる。どの処理系でも1文字の押し戻しを保証する。chの値がEOFのときは、押し戻しは行なわれずに、EOFを返す。

用例 ファイルの先頭から連続した英数字だけをfgetc()で読み込んで表示
非英数字をungetc()で押し戻し、次にfgets()で読み込んで表示

```
#include <stdio.h>
#include <ctype.h>   /* for isalnum() */
int main(void)
{
    FILE *fp;
    int ch;
    char ss[80];
    if ((fp=fopen("tst.txt","w")) == NULL) return 1;    // 書き込み
    fputs("1234abcd#efgh", fp); fclose(fp);
```

```
    if ((fp=fopen("tst.txt", "r")) == NULL) return 1;   // 読み込み
    while ((ch=fgetc(fp)) != EOF) {
        if (isalnum(ch)) putchar(ch);                   // 英数字なら表示
        else { ungetc(ch, fp); break; }                 // そうでないなら戻す
    }
    fgets(ss, 80, fp); printf("¥n%s¥n", ss);
    fclose(fp);
    return 0;
}
```

実行結果

```
1234abcd
#efgh
```

va_start

形式 **#include <stdarg.h>**
void va_start(va_list ap, <最終引数>);

戻値 なし

説明 可変長引数関数のとき可変個引数の先頭へのポインタをapにセットするマクロ。このapは、あとでva_arg()マクロおよびva_end()マクロが使用する。<最終引数>は固定引数の最後のもの。可変個引数並びの「, ...」の直前の引数。

用例 →vprintf()関数(p. 322),「160 可変個引数をもつ関数」(p. 241)

va_arg

形式 **#include <stdarg.h>**
<型> va_arg(va_list ap, <型>);

戻値 取り出した引数

説明 可変長引数関数のときに使うマクロ。apの中から<型>で示すデータをひとつ取り出し。次の実引数の値が順に返されるようにapを更新する。

用例 →「160 可変個引数をもつ関数」(p. 241)

va_end

形式 **#include <stdarg.h>**
void va_end(va_list ap);

戻値 なし

説明 可変長引数関数処理の後始末をして正常な復帰を可能にするマクロ。va_end()を呼び出さないで終了した場合の関数の動作は未定義。

用例 →vprintf()関数(p. 322),「160 可変個引数をもつ関数」(p. 241)

vfprintf

形式 **#include <stdio.h>**
#include <stdarg.h>
int vfprintf(FILE *stream, const char *format, va_list arg);

戻値 正常時：出力した文字数　エラー時：負の値

説明 書式formatと可変個引数情報をもつargを使ってfprintf()と同等の出力を行なう。→vprintf()関数(p. 322), fprintf()関数(p. 292)

vprintf

形式

```
#include <stdio.h>
#include <stdarg.h>
int vprintf(const char *format, va_list arg);
```

戻値 正常時：出力した文字数　エラー時：負の値

説明 書式formatと可変個引数情報をもつargを使ってprintf()と同等の出力を行なう。printf()は可変個引数を必要とするがvprintf()は2個の固定引数だけで利用できる。引数argの中に可変個引数情報を格納してこの関数を呼び出す。va_listはstdarg.hの中で定義されている構造体である。argは使用前にva_startマクロで初期化し、使用後はva_end()マクロで後処理する。→va_startマクロ(p. 321), va_argマクロ(p. 321), va_endマクロ(p. 321)

用例

```
#include <stdio.h>
#include <stdarg.h> /* for va_xxx */
void dataout(char *format, ...);
int main(void)
{
    dataout("d:%d f:%f c:%c¥n", 10, 23.45, 'A');// 出力：d:10 f:23.450000 c:A
    dataout("名前=%s 年齢=%d¥n", "山田", 32);  // 出力：名前=山田 年齢=32
    dataout("Hello¥n");                       // 出力：Hello
    return 0;
}

void dataout(char *format, ...)                // 可変個引数の関数
{
    va_list ap;                                // 可変個引数情報を入れる
    va_start(ap, format);                      // 初期化。最終引数を指定
    vprintf(format, ap);                       // 書式とapを使って実行
    va_end(ap);                                // 後処理
}
```

vsprintf

形式

```
#include <stdio.h>
#include <stdarg.h>
int vsprintf(char *s, const char *format, va_list arg);
```

戻値 正常時：書き込んだ文字数(終端のナル文字は数えない)　エラー時：負の値

説明 書式formatと可変個引数情報をもつargを使ってsprintf()と同等の出力を行なう。→vprintf()関数(p. 322)

wcstombs

形式

```
#include <stdlib.h>
size_t wcstombs(char *mbs, const wchar_t *wcs, size_t n);
```

戻値 多バイト文字列に変換してmbsに書き込んだバイト数を返す
正しい多バイト文字に変換できないとき(size_t)-1を返す

説明 ワイド文字列wcsを多バイト文字列に変換してmbsに格納する。多バイト文字がn文字を超えるときは変換を終了する。

用例 →「157 ワイド文字と多バイト文字の処理1」(p. 234)

wctomb

形式

```
#include <stdlib.h>
int wctomb(char *mbs, wchar_t wch);
```

戻値 ワイド文字に対応する多バイト文字の構成バイト数を返す
wchの値が正しく多バイト文字に変換できない場合は-1をす
mbsがNULLのとき：
　多バイト文字がシフト状態に依存する表現形式であれば：非0
　そうでないなら：0

説明 ワイド文字wchを多バイト文字に変換してmbsに格納する。末尾に'¥0'を補うことはない。格納される文字数は最大MB_CUR_MAXである。

用例 →「157 ワイド文字と多バイト文字の処理1」(p. 234)

INDEX

数字・記号

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

あ行

か行

さ行

●た行

●な行

●は行

ま行

や行

ら行

本書をお読みになったご感想、ご意見を次のURLからお寄せください。

http://www.sbcr.jp/books/hayashi/

本書に関するサポート情報やお問い合わせ受け付けフォームも掲載予定ですので、
あわせてご利用ください。

C言語クイック入門&リファレンス

2010年7月2日　初版第1刷発行

著　者……………………林 晴比古

発行者……………………新田 光敏

発行所……………………ソフトバンク クリエイティブ株式会社

〒107-0052　東京都港区赤坂4-13-13
販売　03(5549)1201
編集　03(5549)1143

本文デザイン…………日野 絵美

本文イラスト…………Suguru.T(NARIKIRI)

カバーデザイン………三部 智也

組　版……………………ホームデータ

印　刷……………………岩岡印刷工業株式会社

Printed in Japan　　ISBN978-4-7973-5999-2

インターネットのサポートページ

林晴比古実用マスターシリーズに関する情報を
インターネットでも配信しています。

http://www.sbcr.jp/books/hayashi/

林 晴比古
実用マスターシリーズ

C言語
クイック入門&リファレンス

9784797359992

1920055026000

ISBN978-4-7973-5999-2

C0055 ¥2600E

定価 本体2,600円 +税

bsearch fputs isxdigit puts strcspn va_start calloc fread labs qsort strerror va_arg ceil free ldexp raise strftime va_end clearerr freopen ldiv rand strlen vfprintf clock frexp localeconv putc strcmp toupper atol fputc isupper putchar strcpy ungetc

林 晴比古 実用マスターシリーズ

C言語クイック入門&リファレンス

林 晴比古

SoftBank Creative